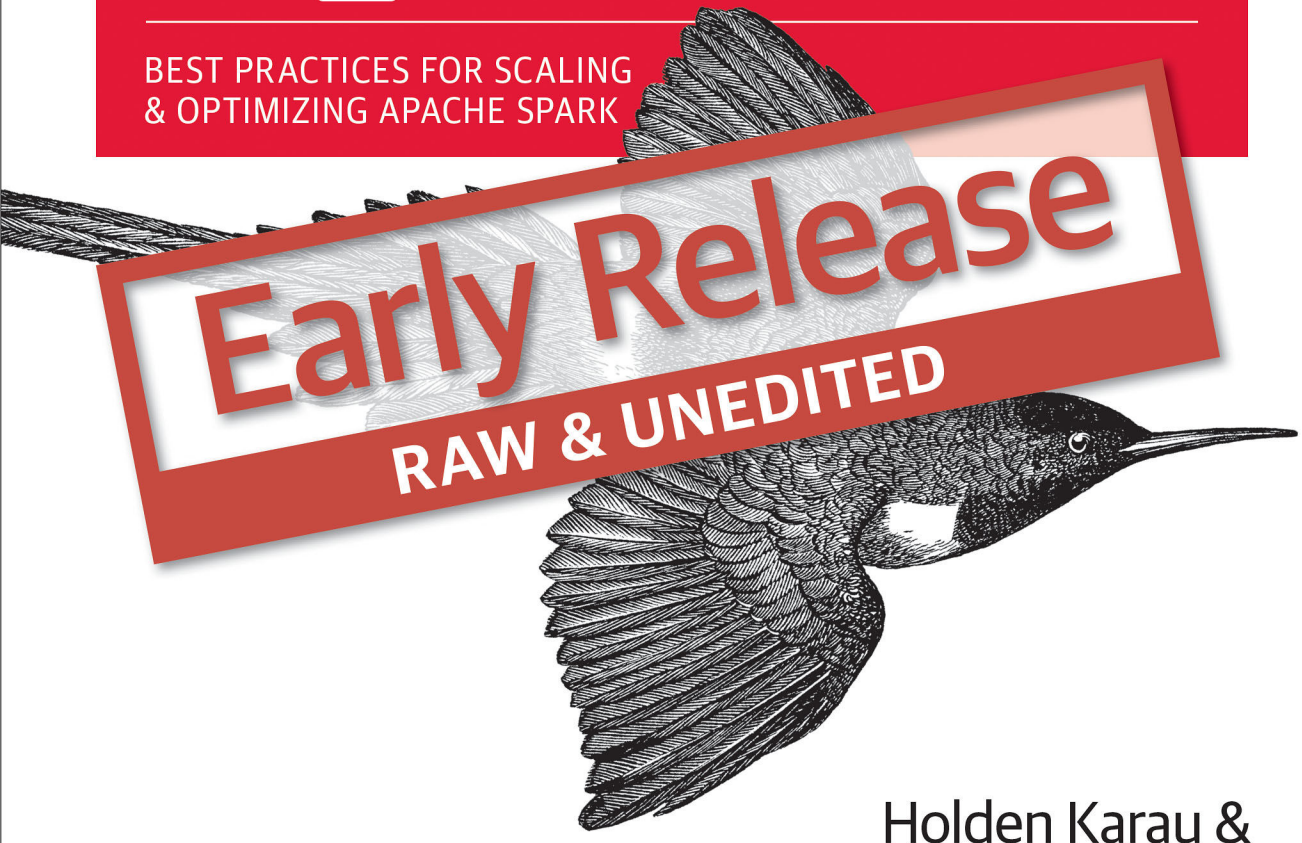
O'REILLY®

High Performance Spark

BEST PRACTICES FOR SCALING & OPTIMIZING APACHE SPARK



Early Release

RAW & UNEDITED

Holden Karau &
Rachel Warren

High Performance Spark

Best Practices for Scaling and Optimizing Apache Spark

Holden Karau and Rachel Warren

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

High Performance Spark
by Holden Karau and Rachel Warren

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Shannon Cutt
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2016: First Edition

Revision History for the First Edition
2016-03-21: First Early Release
2016-07-14: Second Early Release
2016-08-25: Third Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491943205* for release details.

978-1-491-94320-5

[FILL IN]

Table of Contents

Preface

Who Is This Book For?

This book is for data engineers and data scientists who are looking to get the most out of Spark. If you've been working with Spark and invested in Spark but your experience so far has been mired by memory errors and mysterious, intermittent failures, this book is for you. If you have been using Spark for some exploratory work or experimenting with it on the side but haven't felt confident enough to put it into production, this book may help. If you are enthusiastic about Spark but haven't seen the performance improvements from it that you expected, we hope this book can help. This book is intended for those who have some working knowledge of Spark and may be difficult to understand for those with little or no experience with Spark or distributed computing. For recommendations of more introductory literature see "Supporting Books & Materials" on page viii.

We expect this text will be most useful to those who care about optimizing repeated queries in production, rather than to those who are doing primarily exploratory work. While writing highly performant queries is perhaps more important to the data engineer, writing those queries with Spark, in contrast to other frameworks, requires a good knowledge of the data, usually more intuitive to the data scientist. Thus it may be more useful to a data engineer who may be less experienced with thinking critically about the statistical nature, distribution, and layout of your data when considering performance. We hope that this book will help data engineers think more critically about their data as they put pipelines into production. We want to help our readers ask questions such as: "How is my data distributed?", "Is it skewed?", "What is the range of values in a column?", "How do we expect a given value to group?", and "Is it skewed?". And to apply the answers to those questions to the logic of their Spark queries.

However, even for data scientists using Spark mostly for exploratory purposes, this book should cultivate some important intuition about writing performant Spark queries, so that as the scale of the exploratory analysis inevitably grows, you may have

a better shot of getting something to run the first time. We hope to guide data scientists, even those who are already comfortable thinking about data in a distributed way, to think critically about how their programs are evaluated, empowering them to explore their data more fully, more quickly, and to communicate effectively with anyone helping them put their algorithms into production.

Regardless of your job title, it is likely that the amount of data with which you are working is growing quickly. Your original solutions may need to be scaled, and your old techniques for solving new problems may need to be updated. We hope this book will help you leverage Apache Spark to tackle new problems more easily and old problems more efficiently.

Early Release Note

You are reading an early release version of High Performance Spark, and for that, we thank you! If you find errors, mistakes, or have ideas for ways to improve this book, please reach out to us at *high-performance-spark@googlegroups.com*. If you wish to be included in a "thanks" section in future editions of the book, please include your preferred display name.

This is an early release. While there are always mistakes and omissions in technical books, this is especially true for an early release book.

Supporting Books & Materials

For data scientists and developers new to Spark, *Learning Spark* by Karau, Konwinski, Wendel, and Zaharia is an excellent introduction, [1] and "Advanced Analytics with Spark" by Sandy Ryza, Uri Laserson, Sean Owen, Josh Wills is a great book for interested data scientists.

Beyond books, there is also a collection of intro-level Spark training material available. For individuals who prefer video, Paco Nathan has an excellent introduction video series on O'Reilly. Commercially, Databricks as well as Cloudera and other Hadoop/Spark vendors offer Spark training. Previous recordings of Spark camps, as well as many other great resources, have been posted on the Apache Spark documentation page.

1 albeit we may be biased

If you don't have experience with Scala, we do our best to convince you to pick up Scala in Chapter 1, and if you are interested in learning, "Programming Scala, 2nd Edition" by Dean Wampler, Alex Payne is a good introduction.[2]

Conventions Used in this Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

2 Although it's important to note that some of the practices suggested in this book are not common practice in Spark code.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download from the High Performance Spark GitHub Repository, and some of the testing code is available at the "Spark Testing Base" Github Repository. and the Spark Validator Repo.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. The code is also available under an Apache 2 License. Incorporating a significant amount of example code from this book into your product's documentation may require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use *Safari Books Online* as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact the Authors

For feedback on the early release, e-mail us at *high-performance-spark@googlegroups.com*. For random ramblings, occasionally about Spark, follow us on twitter:

Holden: *http://twitter.com/holdenkarau*

Rachel: *https://twitter.com/warre_n_peace*

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

The authors would like to acknowledge everyone who has helped with comments and suggestions on early drafts of our work. Special thanks to Anya Bida and Jakob Odersky for reviewing early drafts and diagrams. We'd also like to thank Mahmoud Hanafy for reviewing and improving the sample code as well as early drafts. We'd also like to thank Michael Armbrust for reviewing and providing feedback on early drafts of the SQL chapter. Justin Pihony has been one of the most active early readers, suggesting fixes in every respect (language, formatting, etc.).

We'd also like to thank all of the readers of our O'Reilly early release who have provided feedback on various errata including Kanak Kshetri.

We'd also like to thank our respective employers for being understanding as we've worked on this book. Especially Lawrence Spracklen who insisted we mention him here :p.

CHAPTER 1

Introduction to High Performance Spark

This chapter provides an overview of what we hope you will be able to learn from this book and does its best to convince you to learn Scala. Feel free to skip ahead to Chapter 2 if you already know what you're looking for and use Scala (or have your heart set on another language).

Spark Versions

Spark follows semantic versioning with the standard [MAJOR].[MINOR].[MAINTENANCE] with API stability for public non-experimental non-developer APIs within minor and maintenance releases. Many of these experimental components are some of the more exciting from a performance standpoint, including `Datasets` — Spark SQL's new structured, strongly-typed, data abstraction. Spark also tries for binary API compatibility between releases, using MiMa [1]; so if you are using the stable API you generally should not need to recompile to run our job against a new version of Spark unless the major version has changed.

This book is created using the Spark 1.6 APIs (and the final version will be updated to 2.0) - but much of the code will work in earlier versions of Spark as well. In places where this is not the case we have attempted to call that out.

1 MiMa is the Migration Manager for Scala and tries to catch binary incompatibilities between releases.

What is Spark and Why Performance Matters

Apache Spark is a high-performance, general-purpose distributed computing system that has become the most active Apache open-source project, with more than 800 active contributors. [2] Spark enables us to process large quantities of data, beyond what can fit on a single machine, with a high-level, relatively easy-to-use API. Spark's design and interface are unique, and it is one of the fastest systems of its kind. Uniquely, Spark allows us to write the logic of data transformations and machine learning algorithms in a way that is parallelizable, but relatively system agnostic. So it is often possible to write computations which are fast for distributed storage systems of varying kind and size.

However, despite its many advantages and the excitement around Spark, the simplest implementation of many common data science routines in Spark can be much slower and much less robust than the best version. Since the computations we are concerned with may involve data at a very large scale, the time and resources that gains from tuning code for performance are enormous. Performance that does not just mean run faster; often at this scale it means getting something to run at all. It is possible to construct a Spark query that fails on gigabytes of data but, when refactored and adjusted with an eye towards the structure of the data and the requirements of the cluster succeeds on the same system with terabytes of data. In the author's experience, writing production Spark code, we have seen the same tasks, run on the same clusters, run 100x faster using some of the optimizations discussed in this book. In terms of data processing, time is money, and we hope this book pays for itself through a reduction in data infrastructure costs and developer hours.

Not all of these techniques are applicable to every use case. Especially because Spark is highly configurable, but also exposed at a higher level than other computational frameworks of comparable power, we can reap tremendous benefits just by becoming more attuned to the shape and structure of your data. Some techniques can work well on certain data sizes or even certain key distributions but not all. The simplest example of this can be how for many problems, using `groupByKey` in Spark can very easily cause the dreaded out of memory exceptions, but for data with few duplicates this operation can be almost the same. Learning to understand your particular use case and system and how Spark will interact with it is a must to solve the most complex data science problems with Spark.

2 From *http://spark.apache.org/* "Since 2009, more than 800 developers have contributed to Spark".

What You Can Expect to Get from This Book

Our hope is that this book will help you take your Spark queries and make them faster, able to handle larger data sizes, and use fewer resources. This book covers a broad range of tools and scenarios. You will likely pick up some techniques which might not apply to the problems you are working with, but which might apply to a problem in the future and which may help shape your understanding of Spark more generally. The chapters in this book are written with enough context to allow the book to be used as a reference; however, the structure of this book is intentional and reading the sections in order should give you not only a few scattered tips but a comprehensive understanding of Apache Spark and how to make it sing.

It's equally important to point out what you will likely not get from this book. This book is not intended to be an introduction to Spark or Scala; several other books and video series are available to get you started. The authors may be a little biased in this regard, but we think "Learning Spark" by Karau, Konwinski, Wendel, and Zaharia as well as Paco Nathan's introduction video series are excellent options for Spark beginners. While this book is focused on performance, it is not an operations book, so topics like setting up a cluster and multi-tenancy are not covered. We are assuming that you already have a way to use Spark in your system and won't provide much assistance in making higher-level architecture decisions. There are future books in the works, by other authors, on the topic of Spark operations that may be done by the time you are reading this one. If operations are your show, or if there isn't anyone responsible for operations in your organization, we hope those books can help you.

Why Scala?

In this book, we will focus on Spark's Scala API and assume a working knowledge of Scala. Part of this decision is simply in the interest of time and space; we trust readers wanting to use Spark in another language will be able to translate the concepts used in this book without presenting the examples in Java and Python. More importantly, it is the belief of the authors that "serious" performant Spark development is most easily achieved in Scala. To be clear these reasons are very specific to using Spark with Scala; there are many more general arguments for (and against) Scala's applications in other contexts.

To Be a Spark Expert You Have to Learn a Little Scala Anyway

Although Python and Java are more commonly used languages, learning Scala is a worthwhile investment for anyone interested in delving deep into Spark development. Spark's documentation can be uneven. However, the readability of the codebase is world-class. Perhaps more than with other frameworks, the advantages of cultivating a sophisticated understanding of the Spark code base is integral to the advanced Spark user. Because Spark is written in Scala, it will be difficult to interact with the

Spark source code without the ability, at least, to read Scala code. Furthermore, the methods in the RDD class closely mimic those in the Scala collections API. RDD functions, such as `map`, `filter`, `flatMap`, `reduce`, and `fold`, have nearly identical specifications to their Scala equivalents [3]. Fundamentally Spark is a functional framework, relying heavily on concepts like immutability and lambda definition, so using the Spark API may be more intuitive with some knowledge of functional programming.

The Spark Scala API is Easier to Use Than the Java API

Once you have learned Scala, you will quickly find that writing Spark in Scala is less painful than writing Spark in Java. First, writing Spark in Scala is significantly more concise than writing Spark in Java since Spark relies heavily on inline function definitions and lambda expressions, which are much more naturally supported in Scala (especially before Java 8). Second, the Spark shell can be a powerful tool for debugging and development, and is only available in languages with existing REPLs (Scala, Python, and R).

Scala is More Performant Than Python

It can be attractive to write Spark in Python, since it is easy to learn, quick to write, interpreted, and includes a very rich set of data science tool kits. However, Spark code written in Python is often slower than equivalent code written in the JVM, since Scala is statically typed, and the cost of JVM communication (from Python to Scala) can be very high. Last, Spark features are generally written in Scala first and then translated into Python, so to use cutting edge Spark functionality, you will need to be in the JVM; Python support for MLlib and Spark Streaming are particularly behind.

Why Not Scala?

There are several good reasons, to develop with Spark in other languages. One of the more important constant reason is developer/team preference. Existing code, both internal and in libraries, can also be a strong reason to use a different language. Python is one of the most supported languages today. While writing Java code can be clunky and sometimes lag slightly in terms of API, there is very little performance cost to writing in another JVM language (at most some object conversions). [4]

3 Although, as we explore in this book, the performance implications and evaluation semantics are quite different.

4 Of course, in performance, every rule has its exception. `mapPartitions` in Spark 1.6 and earlier in Java suffers some severe performance restrictions we discuss in "Iterator-to-Iterator Transformations with `mapPartitions`" on page 106.

While all of the examples in this book are presented in Scala for the final release, we will port many of the examples from Scala to Java and Python where the differences in implementation could be important. These will be available (over time) at our Github. If you find yourself wanting a specific example ported please either e-mail us or create an issue on the github repo.

Spark SQL does much to minimize the performance difference when using a non-JVM language. ??? looks at options to work effectively in Spark with languages outside of the JVM, including Spark's supported languages of Python and R. This section also offers guidance on how to use Fortran, C, and GPU specific code to reap additional performance improvements. Even if we are developing most of our Spark application in Scala, we shouldn't feel tied to doing everything in Scala, because specialized libraries in other languages can be well worth the overhead of going outside the JVM.

Learning Scala

If after all of this we've convinced you to use Scala, there are several excellent options for learning Scala. Spark 1.6 is built against Scala 2.10 and cross-compiled against Scala 2.11, and Spark 2.0 will be built against Scala 2.11 and possibly cross compiled against Scala 2.10 and or 2.12. Depending on how much we've convinced you to learn Scala, and what your resources are, there are a number of different options ranging from books to MOOCs to professional training.

For books, *Programming Scala, 2nd Edition* by Dean Wampler and Alex Payne can be great, although much of the actor system references are not relevant while working in Spark. The Scala language website also maintains a list of Scala books.

In addition to books focused on Spark, there are online courses for learning Scala. *Functional Programming Principles in Scala*, taught by Martin Ordersky, its creator, is on Coursera as well as Introduction to Functional Programming on edX. A number of different companies also offer video-based Scala courses, none of which the authors have personally experienced or recommend.

For those who prefer a more interactive approach, professional training is offered by a number of different companies including, Lightbend (formerly Typesafe). While we have not directly experienced Typesafe training, it receives positive reviews and is known especially to help bring a team or group of individuals up to speed with Scala for the purposes of working with Spark.

Conclusion

Although you will likely be able to get the most out of Spark performance if you have an understanding of Scala, working in Spark does not require a knowledge of Scala.

For those whose problems are better suited to other languages or tools, techniques for working with other languages will be covered in ???. This book is aimed at individuals who already have a grasp of the basics of Spark, and we thank you for choosing *High Performance Spark* to deepen your knowledge of Spark. The next chapter will introduce some of Spark's general design and evaluation paradigms which are important to understanding how to efficiently utilize Spark.

CHAPTER 2
How Spark Works

This chapter introduces Spark's place in the big data ecosystem and its overall design. Spark is often considered an alternative to Apache MapReduce, since Spark can also be used for distributed data processing with Hadoop. [1], packaged with the distributed file system, Apache Hadoop Distributed File System.] As we will discuss in this chapter, Spark's design principals are quite different from MapReduce's and unlike Hadoop MapReduce, Spark does not need to be run in tandem with Apache Hadoop - although it often is. Spark has inherited parts of its API, design, and supported formats from other existing computational frameworks, particularly DrayadLINQ. However, Spark's internals, especially how it handles failures, differ from many traditional systems. [2] Spark's ability to leverage lazy evaluation within memory computations make it particularly unique. Spark's creators believe it to be the first high-level programing language for fast, distributed data processing. [3] Understanding the general design principals behind Spark will be useful for understanding the performance of Spark jobs.

To get the most out of Spark, it is important to understand some of the principles used to design Spark and, at a cursory level, how Spark programs are executed. In this

1 MapReduce is a programmatic paradigm that defines programs in terms of *map* procedures that filter and sort data onto the nodes of a distributed system, and *reduce* procedures that aggregate the data on the mapper nodes. Implementations of MapReduce have been written in many languages, but the term usually refers to a popular implementation called link::http://hadoop.apache.org/[*Hadoop MapReduce*

2 DryadLINQ is a Microsoft research project that puts the .NET Language Integrated Query (LINQ) on top of the Dryad distributed execution engine. Like Spark, The DrayadLINQ API defines an object representing a distributed dataset and exposes functions to transform data as methods defined on the dataset object. DrayadLINQ is lazily evaluated and its scheduler is similar to Spark's. However DrayadLINQ doesn't use in-memory storage. For more information see the DrayadLINQ documentation.

3 See the original Spark Paper.

chapter, we will provide a broad overview of Spark's model of parallel computing and a thorough explanation of the Spark scheduler and execution engine. We will refer to the concepts in this chapter throughout the text. Further, we hope this explanation will provide you with a more precise understanding of some of the terms you've heard tossed around by other Spark users and in the Spark documentation.

How Spark Fits into the Big Data Ecosystem

Apache Spark is an open source framework that provides highly generalizable methods to process data in parallel. On its own, Spark is not a data storage solution. Spark can be run locally, on a single machine with a single JVM (called local mode). More often Spark is used in tandem with a distributed storage system to write the data processed with Spark (such as HDFS, Cassandra, or S3) and a cluster manager to manage the distribution of the application across the cluster. Spark currently supports three kinds of cluster managers, Standalone Cluster Manager, Apache Mesos, and Hadoop YARN. The Standalone Cluster Manager is included in Spark and requires Spark to be installed in each node of a cluster.

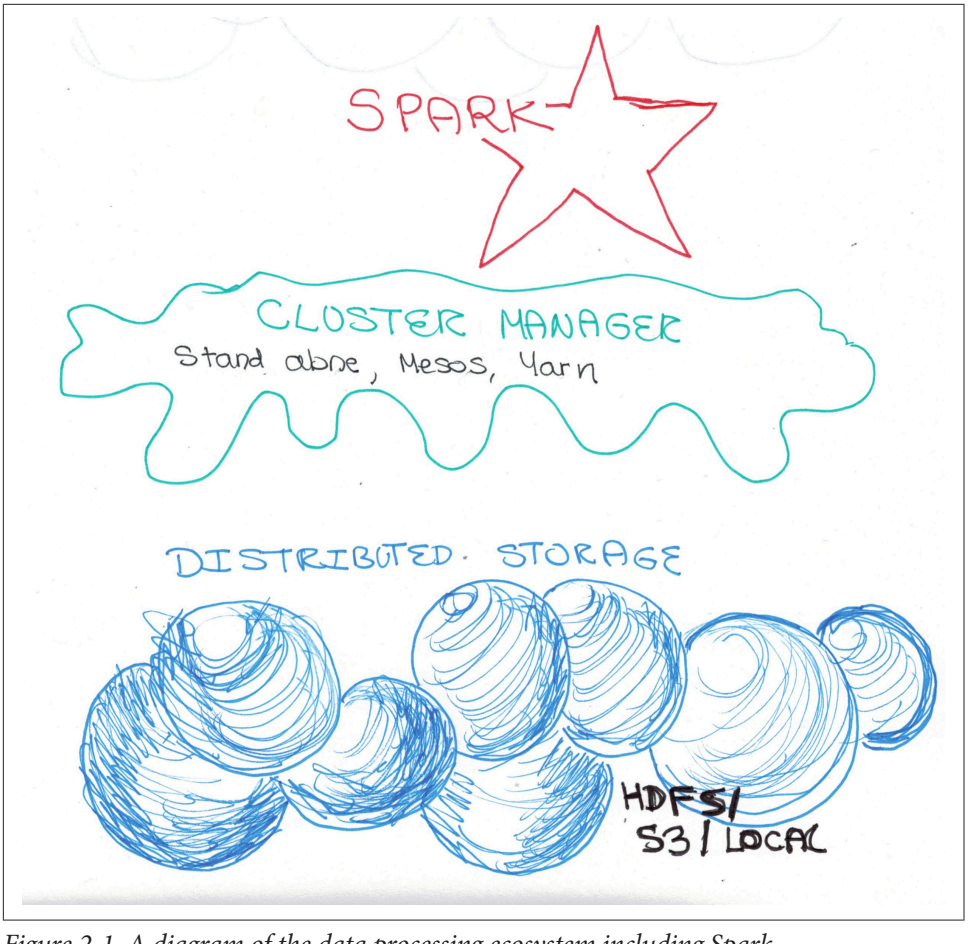


Figure 2-1. A diagram of the data processing ecosystem including Spark.

Spark Components

Spark provides a high-level query language to process data. Spark Core, the main data processing framework in the Spark ecosystem, has APIs in Scala, Java, Python, and R. Spark is built around a data abstraction called *Resilient Distributed Datasets* known as RDDs . RDDs are a representation of lazily evaluated, statically typed, distributed collections. RDDs have a number of predefined "coarse grained" transformations (transformations that are applied to the entire dataset), such as `map`, `join`, and `reduce`, as well as I/O functionality, to move data in and out of storage or back to the driver.

While Spark also supports R, at present the RDD interface is not available.

In addition to Spark Core, the Spark ecosystem includes a number of other first-party components for more specific data processing tasks, including Spark SQL, Spark MLLib, Spark ML, Spark Streaming, and GraphX [4]. These components have many of the same generic performance considerations as the core. However, some of them have unique considerations - like SQL's different optimizer.

Spark SQL is a component that can be used in tandem with the Spark Core. Spark SQL defines an interface for a semi-structured data type, called `DataFrames` and a typed version called `Datasets`, with APIs in Scala, Java, Python, and R, as well as support for basic SQL queries. Spark SQL is a very important component for Spark performance, and much of what can be accomplished with Spark core can done leveraging Spark SQL, so we cover it deeply in Chapter 3.

Spark has two machine learning packages, ML and MLlib. MLlib, one of Spark's machine learning components is a package of machine learning and statistics algorithms written with Spark. Spark ML is still in the early stages, but since Spark 1.2, it provides a higher-level API than MLlib with the goal of allowing users to more easily create practical machine learning pipelines. Spark MLLib is primarily built on top of RDDs, while ML is build on top of SparkSQL data frames. [5] Eventually the Spark community plans to move over to ML and deprecate MLlib. Spark ML and MLLib have some unique performance considerations, especially when working with large data sizes and caching, and we cover some these in ???.

Spark Streaming uses the scheduling of the Spark Core for streaming analytics on mini batches of data. Spark Streaming has a number of unique considerations, such as the window sizes used for batches. We offer some tips for using Spark Streaming in ???.

GraphX is a graph processing framework built on top of Spark with an API for graph computations. GraphX is one of the least mature components of Spark, so we don't cover it in much detail. In future versions of Spark, typed graph functionality will be introduced on top of the Dataset API. We will provide a cursory glance at GraphX in ???.

4 Although GraphX is not actively developed at this point, and will likely be replaced with Graph frames or similar

5 See The MLlib documentation.

This book will focus on optimizing programs written with the Spark Core and Spark SQL. However, since MLLib and the other frameworks are written using the Spark API, this book will provide the tools you need to leverage those frameworks more efficiently. Who knows, maybe by the time you're done, you will be ready to start contributing your own functions to MLlib and ML!

Beyond first party components, a large number of libraries both extend Spark for different domains and offer tools to connect it to different data sources. Many libraries are listed at *http://spark-packages.org/*, and can be dynamically included at runtime with `spark-submit` or the `spark-shell` and added as build dependencies to your `maven` or `sbt` project. We first use Spark packages to add support for csv data in "Additional Formats" on page 68 and then in more detail in ???

Spark Model of Parallel Computing: RDDs

Spark allows users to write a program for the *driver* (or master node) on a cluster computing system that can perform operations on data in parallel. Spark represents large datasets as RDDs, immutable, distributed collections of objects, which are stored in the *executors* (or slave nodes). The objects that comprise RDDs are called partitions and may be (but do not need to be) computed on different nodes of a distributed system. The Spark cluster manager handles starting and distributing the Spark executors across a distributed system according to the configuration parameters set by the Spark application. The Spark execution engine itself distributes data across the executors for a computation. See Figure 2-4.

Rather than evaluating each transformation as soon as specified by the driver program, Spark evaluates RDDs lazily, computing RDD transformations only when the final RDD data needs to be computed (often by writing out to storage or collecting an aggregate to the driver). Spark can keep an RDD loaded in-memory on the executor nodes throughout the life of a Spark application for faster access in repeated computations. As they are implemented in Spark, RDDs are immutable, so transforming an RDD returns a new RDD rather than the existing one. As we will explore in this chapter, this paradigm of lazy evaluation, in-memory storage and immutability allows Spark to be easy-to-use, fault-tolerant and generally highly performant.

Lazy Evaluation

Many other systems for in-memory storage are based on "fine grained" updates to mutable objects, i.e., calls to a particular cell in a table by storing intermediate results. In contrast, evaluation of RDDs is completely lazy. Spark does not begin computing the partitions until an action is called. An action is a Spark operation which returns something other than an RDD, triggering evaluation of partitions and possibly returning some output to a non-Spark system, for example bringing data back to the driver (with operations like `count` or `collect`) or writing data to external storage

storage system (such as `copyToHadoop`). Actions trigger the scheduler, which builds a *directed acyclic graph* (called the DAG), based on the dependencies between RDD transformations. In other words, Spark evaluates an action by working backward to define the series of steps it has to take to produce each object in the final distributed dataset (each partition). Then, using this series of steps called the execution plan, the scheduler computes the missing partitions for each stage until it computes the result.

Not all transformations are 100% lazy, `sortByKey` needs to evaluate the RDD to determine the range of data - so it involves both a transformation and an action.

Performance & Usability Advantages of Lazy Evaluation

Lazy evaluation allows Spark to combine operations that don't require communication with the driver (called transformations with one-to-one dependencies) to avoid doing multiple passes through the data. For example, suppose a Spark program calls a `map` and a `filter` function on the same RDD. Spark can look at each record once and compute both the `map` and the `filter` for the records on each partition in the executor nodes, rather than doing two passes through the data, one for the `map` and one for the `filter`, theoretically reducing the computational complexity by half.

Spark's lazy evaluation paradigm is not only more efficient, it is also easier to implement the same logic in Spark than in a different framework, like MapReduce that require the developer to do the work to consolidate her mapping operations. Spark's clever lazy evaluation strategy lets us be lazy and express the same logic in far fewer lines of code, because we can chain together operations with narrow dependencies and let the Spark evaluation engine do the work of consolidating them.

Consider the classic word count example which, given a dataset of documents, parses the text into words and then computes the count for each word. The Apache docs provide a word count example, which even in its simplest form is roughly fifty lines of code (excluding import statements) in Java. A comparable Spark implementation is roughly fifteen lines of code in Java and five in Scala, available on the Apache website. For reference this is a fast routine to compute word count (excluding reading in the data Excluding the steps to read in the data mapping documents to words and counting the words can be expressed in Spark as follows.

Example 2-1.

```
def simpleWordCount(rdd: RDD[String]): RDD[(String, Int)] = {
  val words = rdd.flatMap(_.split(" "))
  val wordPairs = words.map((_, 1))
  val wordCounts = wordPairs.reduceByKey(_ + _)
```

```
    wordCounts
  }
```

Furthermore if we were to filter out some "stop words" and punctuation from each document before computing the word count, this would require adding the filter logic to the mapper to avoid doing a second pass through the data. An implementation of this routine for MapReduce can be found here: *https://github.com/kite-sdk/kite/wiki/WordCount-Version-Three*. In contrast, we can modify the Spark routine above by simply putting a filter step before we begin the code shown above and Spark's lazy evaluation will consolidate the map and filter steps for us. .Word count example with stop words.

Example 2-2.

```
def withStopWordsFiltered(rdd : RDD[String], illegalTokens : Array[Char],
  stopWords : Set[String]): RDD[(String, Int)] = {
  val seperators = illegalTokens ++ Array[Char](' ')
  val tokens: RDD[String] = rdd.flatMap(_.split(seperators).
    map(_.trim.toLowerCase))
  val words = tokens.filter(token =>
    !stopWords.contains(token) && (token.length > 0) )
  val wordPairs = words.map((_, 1))
  val wordCounts = wordPairs.reduceByKey(_ + _)
  wordCounts
}
```

Lazy Evaluation & Fault Tolerance

Spark is fault-tolerant, because each partition of the data contains the dependency information needed to re-calculate the partition. Distributed systems, based on mutable objects and strict evaluation paradigms, provide fault tolerance by logging updates or duplicating data across machines. In contrast, Spark does not need to maintain a log of updates to each RDD or log the actual intermediary steps, since the RDD itself contains all the dependency information needed to replicate each of its partitions. Thus, if a partition is lost, the RDD has enough information about its lineage to recompute it, and that computation can be parallelized to make recovery faster.

In-Memory Storage and Memory Management

Spark's biggest performance advantage over MapReduce is in use cases involving repeated computations. Much of this performance increase is due to Spark's storage system. Rather than writing to disk between each pass through the data, Spark has the option of keeping the data on the executors loaded into memory. That way, the data on each partition is available in-memory each time it needs to be accessed.

Spark offers three options for memory management: in-memory as deserialized data, in-memory as serialized data, and on disk. Each has different space and time advantages.

1. *In memory as deserialized Java objects:* The most intuitive way to store objects in RDDs is as the original deserialized Java objects that are defined by the driver program. This form of in-memory storage is the fastest, since it reduces serialization time; however, it may not be the most memory efficient, since it requires the data to be stored as objects.
2. *As serialized data:* Using the Java serialization library, Spark objects are converted into streams of bytes as they are moved around the network. This approach may be slower, since serialized data is more CPU-intensive to read than deserialized data; however, it is often more memory efficient, since it allows the user to choose a more efficient representation. While Java serialization is more efficient than full objects, ??? serialization can be even more space efficient.
3. *On Disk:* RDDs, whose partitions are too large to be stored in RAM on each of the executors, can be written to disk. This strategy is obviously slower for repeated computations, but can be more fault-tolerant for long strings of transformations and may be the only feasible option for enormous computations.

The `persist()` function in the RDD class lets the user control how the RDD is stored. By default, `persist()` stores an RDD as deserialized objects in memory, but the user can pass one of numerous storage options to the `persist()` function to control how the RDD is stored. We will cover the different options for RDD reuse in "Types of Reuse: Cache, Persist, Checkpoint, Shuffle Files" on page 120. When persisting RDDs, the default implementation of RDDs evicts the least recently used partition (called LRU caching). However you can change this behavior and control Spark's memory prioritization with the `persistencePriority()` function in the RDD class. See "LRU Caching" on page 125.

Immutability and the RDD Interface

Spark defines an RDD interface with the properties that each type of RDD must implement. These properties include the RDD's dependencies and information about data locality that are needed for the execution engine to compute that RDD. Since RDDs are statically typed and immutable, calling a transformation on one RDD will not modify the original RDD but rather return a new RDD object with a new definition of the RDD's properties.

RDDs can be created in two ways: (1) by transforming an existing RDD or (2) from a Spark Context, `SparkContext`. The Spark Context represents the connection to a Spark cluster and one running Spark application. The Spark Context can be used to create an RDD from a local Scala object (using the `makeRDD`, or `parallelize` meth-

ods) or by reading from stable storage (text files, binary files, a Hadoop Context, or a Hadoop file). DataFrames and Datasets can be read using the Spark SQL equivalent to a Spark Context, the `SQLContext` (soon to be renamed).

Spark uses five main properties to represent an RDD internally. The three required properties are the list of partition objects, a function for computing an iterator of each partition, and a list of dependencies on other RDDs. Optionally, RDDs also include a partitioner (for RDDs of rows of key-value pairs represented as Scala tuples) and a list of preferred locations (for the HDFS file). Although, as an end user, you will rarely need these five properties and are more likely to use predefined RDD transformations, it is helpful to understand the properties and know how to access them for debugging and for a better conceptual understanding. These five properties correspond to the following five methods available to the end user (you):

- `partitions()` Returns an array of the partition objects that make up the parts of the distributed dataset. In the case of an RDD with a partitioner, the value of the index of each partition will correspond to the value of the getPartition function for each key in the data associated with that partition.
- `iterator(p, parentIters)` Computes the elements of partition p given iterators for each of its parent partitions. This function is called in order to compute each of the partitions in this RDD. This is not intended to be called directly by the user, rather this is used by Spark when computing actions. Still, referencing the implementation of this function can be useful in determining how each partition of an RDD transformation is evaluated.
- `dependencies()` Returns a sequence of dependency objects. The dependencies let the scheduler know how this RDD depends on other RDDs. There are two kinds of dependencies: *Narrow Dependencies* (`NarrowDependency` objects), which represent partitions that depend on one or a small subset of partitions in the parent, and *Wide Dependencies* (`ShuffleDependency` objects), which are used when a partition can only be computed by rearranging all the data in the parent. We will discuss the types of dependencies in "Wide vs. Narrow Dependencies" on page 29.
- `partitioner()` Returns a Scala option type of a `partitioner` object if the RDD has a function between datapoint and partitioner associated with it, such as a `hashPartitioner`. This function returns `None` for all RDDs that are not of type tuple (do not represent Key-value data). An RDD that represents an HDFS file (implemented in `NewHadoopRDD.scala`) has a partitioner for each block of the file. We will discuss partitioning in detail in "Using the Spark Partitioner Object" on page 149.
- `preferredLocations(p)` Returns information about the data locality of a partition, p. Specifically, this function returns a sequence of strings representing some

information about each of the nodes, where the split `p` is stored. In an RDD representing an HDFS file, each string in the result of `preferedLocations` is the Hadoop name of the node.

Types of RDDs

The implementation of the Spark Scala API contains an abstract class, `RDD`, which contains not only the five core functions of RDDs, but also those transformations and actions that are available to all RDDs, such as `map` and `collect`. Functions defined only on RDDs of a particular type are defined in several RDD Functions classes, including `PairRDDFunctions`, `OrderedRDDFunctions` and `GroupedRDDFunctions`. The additional methods in these classes are made available by implicit conversion from the abstract RDD class, based on type information or when a transformation is applied to an RDD.

The Spark API also contains specific implementations of the RDD class that define specific behavior by overriding the core properties of the RDD. These include the `NewHadoopRDD` class discussed above, which represents an RDD created from an HDFS file system, and `ShuffledRDD`, which represents an RDD, that was already partitioned. Each of these RDD implementations contains functionality that is specific to RDDs of that type. Creating an RDD, either through a transformation or from a Spark Context, will return one of these implementations of the RDD class. Some RDD operations have a different signature in Java than in Scala. These are defined in the `JavaRDD.java` class.

Find out what type of an RDD is using the `toDebugString` function which is defined on all RDDs. This will tell you both what kind of RDD you have an provide a list of its parent RDDs. See ??? for a complete example.

We will discuss the different types of RDDs and RDD transformations in detail in Chapter 5 and Chapter 6

Functions on RDDs: Transformations vs. Actions

There are two types of functions defined on RDDs, *actions* and *transformations*. Actions are functions that return something that is not an RDD (including a side effect) and transformations are functions that return another RDD.

Each Spark program must contain an action, since actions either bring information back to the driver or write the data to stable storage; actions are what force evaluation

of a Spark program. Actions that bring data back to the driver include `collect`, `count`, `collectAsMap`, `sample`, `reduce` and `take`.

Some of these actions do not scale well, since they can cause memory errors in the driver. In general, it is best to use actions like `take`, `count`, and `reduce`, which bring back a fixed amount of data to the driver, rather than `collect` or `sample`.

Actions that write to storage include `saveAsTextFile`, `saveAsSequenceFile`, and `saveAsObjectFile`. Most actions that save to Hadoop are made available only on RDDs of key-value pairs; they are defined both in the `PairRDDFunctions` class (which provides methods for RDDs of tuple type by implicit conversation) and the `NewHadoopRDD` class, which is an implementation for RDDs that were created by reading from Hadoop. Some save functions like, `saveAsTextFile`, and `saveAsObjectFile`, are available on all RDDs and work by adding an implicit null key to each record (which is the ignored by the saving level). Functions that return nothing (or `Unit` in Scala), such as `foreach`, are also actions and force execution of a Spark job. `forEach` can be used to force evaluation of an RDD, but is also often used to write out to non-supported formats (like web endpoints).

Most of the power of the Spark API is in its transformations. Spark transformations are general coarse grained transformations used to sort, reduce, group, sample, filter, and map distributed data. We will talk about transformations in detail in both Chapter 6, which deals exclusively with transformations on RDDs of key / value data, and Chapter 5, and we will talk about advanced performance considerations with respect to data transformations.

Wide vs. Narrow Dependencies

For the purpose of understanding how RDDs are evaluated, the most important thing to know about transformations is that they fall into two categories: transformations with *narrow dependencies* and transformations with *wide dependencies*. The narrow vs. wide distinction has significant implications for the way Spark evaluates a transformation and, consequently, for its performance. We will define narrow and wide transformations for the purpose of understanding Spark's execution paradigm in "Spark Job Scheduling" on page 31 of this chapter, but we will save the longer explanation of the performance considerations associated with them for Chapter 5.

Conceptually, narrow transformations are operations with dependencies on just one or a known set of partitions in the parent RDD which can be determined at design time. Thus narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions. In contrast, transformations with wide dependencies cannot be executed on arbitrary rows and instead require the data

to be partitioned in a particular way. Transformations with wide dependencies include, `sort`, `reduceByKey`, `groupByKey`, `join`, and anything that calls for repartition.

We call the process of moving the records in an RDD to accommodate a partitioning requirement, a *shuffle*. In certain instances, for example, when Spark already knows the data is partitioned in a certain way, operations with wide dependencies do not cause a shuffle. If an operation will require a shuffle to be executed, Spark adds a `ShuffledDependency` object to the dependency list associated with the RDD. In general, shuffles are expensive, and they become more expensive the more data we have, and the greater percentage of that data has to be moved to a new partition during the shuffle. As we will discuss at length in Chapter 6, we can get a lot of performance gains out of Spark programs by doing fewer and less expensive shuffles.

The next two diagrams illustrates the difference in the dependency graph for transformations with narrow vs. transformations with wide dependencies. Figure 2-2 shows narrow dependencies in which each child partition (each of the blue squares on the bottom rows) depends on a known subset of parent partitions (narrow dependencies are shown with blue arrows). The left represents a dependency graph of narrow transformations such as `map`, `filter`, `mapPartitions` and `flatMap`. On the upper right are dependencies between partitions for `coalesce`, a narrow transformation. In this instance we try to illustrate that the child partitions may depend on multiple parent partitions, but that so long as the set of parent partitions can be determined regardless of the values of the data in the partitions, the transformation qualifies as narrow.

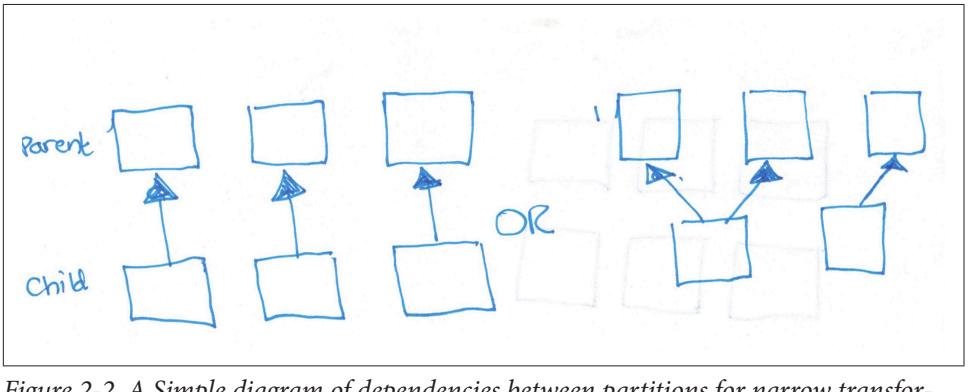


Figure 2-2. A Simple diagram of dependencies between partitions for narrow transformations.

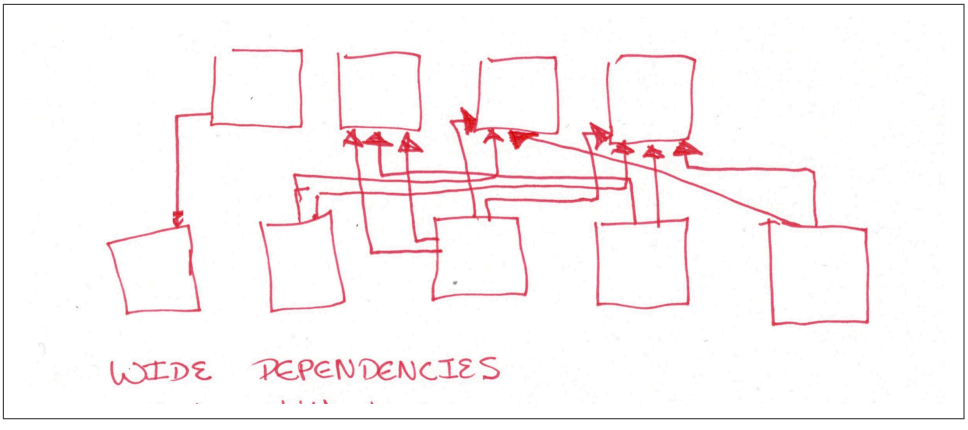


Figure 2-3. A Simple diagram of dependencies between partitions for wide transformations.

The second diagram shows wide dependencies between partitions. In this case the child partitions (shown below) depend on an arbitrary set of parent partitions. The wide dependencies (displayed as red arrows) cannot be known fully before the data is evaluated. In contrast to the `coalesce` operation, data is partitioned according to its value. The dependency graph for any operations that cause a shuffle such as `groupBy Key`, `reduceByKey`, `sort`, and `sortByKey` follows this pattern.

Join is a bit more complicated, since it can have wide or narrow dependencies depending on how the two parent RDDs are partitioned. We illustrate the dependencies in different scenarios for the join operation in "Core Spark Joins" on page 81.

Spark Job Scheduling

A Spark application consists of a driver process, which is where the high-level Spark logic is written, and a series of executor processes that can be scattered across the nodes of a cluster. The Spark program itself runs in the driver node and sends some instructions to the executors. One Spark cluster can run several Spark applications concurrently. The applications are scheduled by the cluster manager and correspond to one SparkContext. Spark applications can run multiple concurrent jobs. Jobs correspond to each action called on an RDD in a given application. In this section, we will describe the Spark application and how it launches Spark jobs, the processes that compute RDD transformations.

Resource Allocation Across Applications

Spark offers two ways of allocating resources across applications: *static allocation* and *dynamic allocation*. With static allocation, each application is allotted a finite maxi-

mum of resources on the cluster and reserves them for the duration of the application (as long as the Spark Context is still running). Within the static allocation category, there are many kinds of resource allocation available, depending on the cluster. For more information, see the Spark documentation for job scheduling.

Since 1.2, Spark offers the option of dynamic resource allocation which expands the functionality of static allocation. In dynamic allocation, executors are added and removed from a Spark application as needed, based on a set of heuristics for estimated resource requirement. We will discuss resource allocation in ???.

The Spark application

A Spark application corresponds to a set of Spark jobs defined by one Spark Context in the driver program. A Spark application begins when a Spark Context is started. When the Spark Context is started, each worker node starts an executor (its own Java Virtual Machine, JVM).

The Spark Context determines how many resources are allotted to each executor, and when a Spark job is launched, each executor has slots for running the tasks needed to compute an RDD. In this way, we can think of one Spark Context as one set of configuration parameters for running Spark jobs. These parameters are exposed in the `SparkConf` object, which is used to create a Spark Context. We will discus how to use the parameters in ???. Often, but not always, applications correspond to users. That is, each Spark program running on your cluster likely uses one Spark Context.

RDDs cannot be shared between applications, so transformations, such as `join` that use more than one RDD must have the same Spark Context.

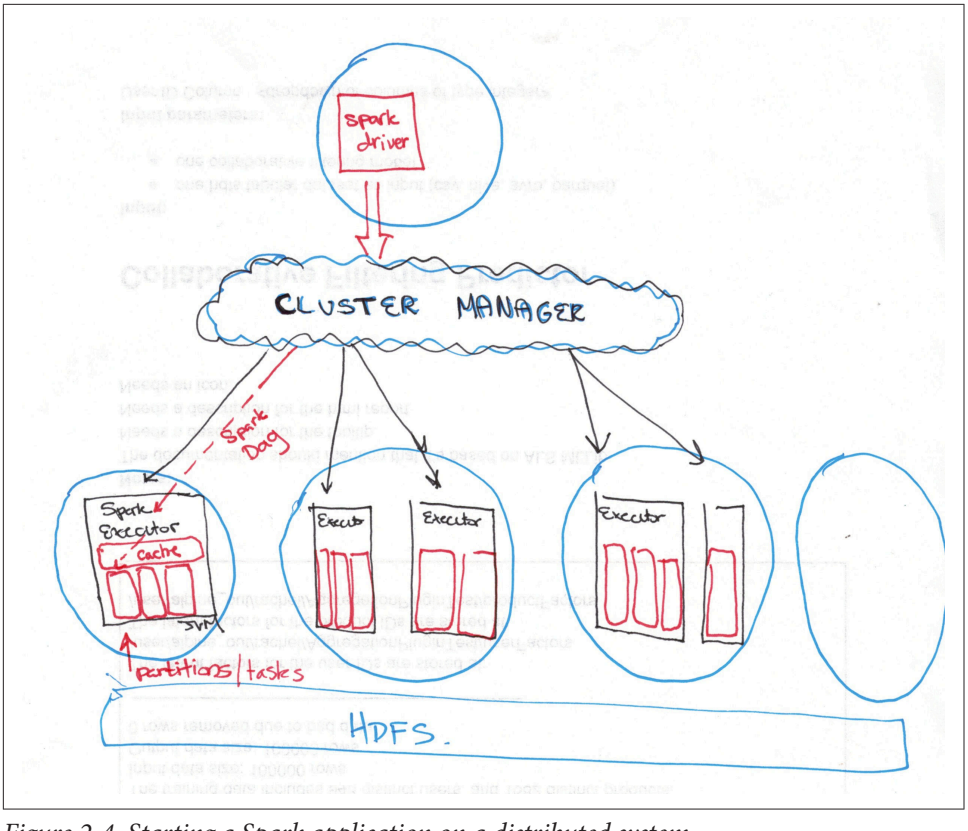


Figure 2-4. Starting a Spark application on a distributed system.

The above diagram illustrates what happens when we start a Spark Context. First The driver program pings the cluster manager. The cluster manager launches a number of Spark executors, JVMs, (shown as black boxes) on the worker nodes of the cluster (shown as blue circles). One node can have multiple Spark executors, but an executor cannot span multiple nodes. An RDD will be evaluated across the executors in partitions (shown as red rectangles). Each executor can have multiple partitions, but a partition cannot be spread across multiple executors.

By default, Spark queues jobs in a first in, first out basis. However, Spark does offer a fair scheduler, which assigns tasks to concurrent jobs in round-robin fashion, i.e. parceling out a few tasks for each job until the jobs are all complete. The fair scheduler ensures that jobs get a more even share of cluster resources. The Spark application then launches jobs in the order that their corresponding actions were called on the Spark Context.

The Anatomy of a Spark Job

In the Spark lazy evaluation paradigm, a Spark application doesn't "do anything" until the driver program calls an action. With each action, the Spark scheduler builds an execution graph and launches a *Spark job*. Each job consists of *stages*, which are steps in the transformation of the data needed to materialize the final RDD. Each stage consists of a collection of *tasks* that represent each parallel computation and are performed on the executors.

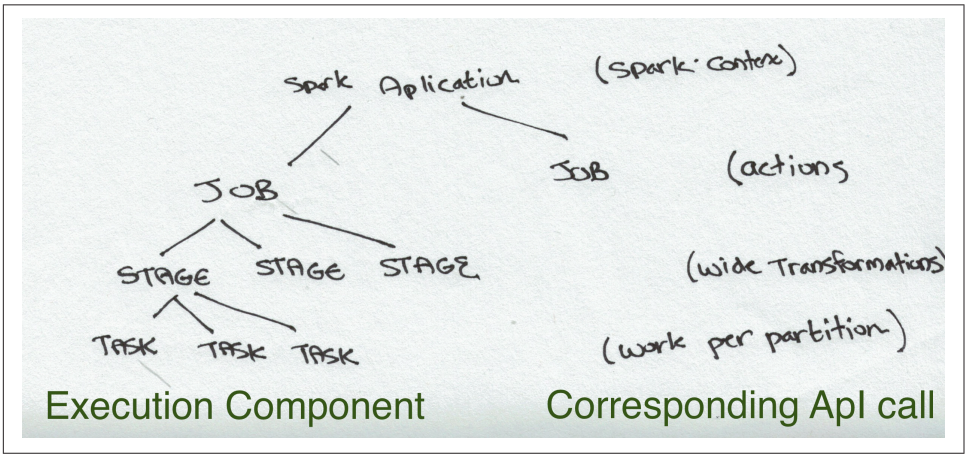


Figure 2-5. The Spark application Tree

The above diagram shows a tree of the different components of a Spark application. An application corresponds to starting a Spark Context. Each application may contain many jobs that correspond to one RDD action. Each job may contain several stages which correspond to each wide transformation. Each stage is composed of one or many tasks which correspond to a parallelizable unit of computation done in each stage. There is one task for each partition in the resulting RDD of that stage.

The DAG

Spark's high-level scheduling layer uses RDD dependencies to build a *Directed Acyclic Graph* (a DAG) of stages for each Spark job. In the Spark API, this is called the `DAG Scheduler`, and as you have probably noticed, errors that have to do with connecting to your cluster, your configuration parameters, or launching a Spark job show up as DAG Scheduler errors. This is because the execution of a Spark job is handled by the DAG. The DAG builds a graph of stages for each job, determines the locations to run each task, and passes that information on to the `TaskScheduler`, which is responsible for running tasks on the cluster.

Jobs

A Spark job is the highest element of Spark's execution hierarchy. Each Spark job corresponds to one action, called by the Spark application. As we discussed in "Functions on RDDs: Transformations vs. Actions" on page 28, one way to conceptualize an action is as something that brings data out of the RDD world of Spark into some other storage system (usually by bringing data to the driver or writing to some stable storage system).

Since the edges of the Spark execution graph are based on dependencies between RDD transformations (as illustrated by Figure 2-2 and Figure 2-3), an operation that returns something other than an RDD cannot have any children. Thus, an arbitrarily large set of transformations may be associated with one execution graph. However, as soon as an action is called, Spark can no longer add to that graph and launches a job including those transformations that were needed to evaluate the final RDD that called the action.

Stages

Recall that Spark lazily evaluates transformations; transformations are not executed until actions are called. As mentioned above, a job is defined by calling an action. The action may include several transformations, and wide transformations define the breakdown of jobs into *stages*.

Each stage corresponds to a `ShuffleDependency` created by a wide transformation in the Spark program. At a high level, one stage can be thought of as the set of computations (tasks) that can each be computed on one executor without communication with other executors or with the driver. In other words, a new stage begins whenever network communication between workers is required, such as in a shuffle. These dependencies that create stage boundaries are called `ShuffleDependencies`, and as we discussed in "Wide vs. Narrow Dependencies" on page 29, they are caused by those wide transformations, such as `sort` or `groupByKey`, which require the data to be re-distributed across the partitions. Several transformations with narrow dependencies can be grouped into one stage. For example, as we saw in the word count example where we filtered stop words, `???`, Spark combines `map` and a `filter` steps into one stage since neither transformation require a shuffle. Thus, each executor can apply the `map` and `filter` steps consecutively in one pass of the data.

The same operations on RDDs with known partitioners and RDDs without a known partitioner can result in different stage boundaries, as the shuffle can often be avoided with a known partitioner.

Because the stage boundaries require communication with the driver, the stages associated with one job generally have to be executed in sequence rather than in parallel. It is possible to execute stages in parallel if they are used to compute different RDDs which are combined in a downstream transformation such as a `join`. However, the wide transformations needed to compute one RDD have to be computed in sequence and thus, it is usually desirable to design your program to require fewer shuffles.

Tasks

A stage consists of tasks. The *task* is the smallest unit in the execution hierarchy, and each can represent one local computation. Each of the tasks in one stage all execute the same code on a different piece of the data. One task cannot be executed on more than one executor. However, each executor has a dynamically allocated number of slots for running tasks and may run many tasks concurrently throughout its lifetime. The number of tasks per stage corresponds to the number of partitions in the output RDD of that stage.

The following diagram shows the evaluation of a Spark job that is the result of a driver program that calls the following simple Spark program:

Example 2-3.

```
def simpleSparkProgram(rdd : RDD[Double]): Long ={
//stage1
  rdd.filter(_< 1000.0)
     .map(x => (x , x) )
//stage2
     .groupByKey()
     .map{ case(value, groups) => (groups.sum, value)}
//stage 3
     .sortByKey()
     .count()
}
```

The stages (black boxes) are bounded by the shuffle operations `groupByKey` and `sortByKey`. Each stage consists of several tasks, one for each partition in the result of the RDD transformations (shown as red squares), which are executed in parallel.

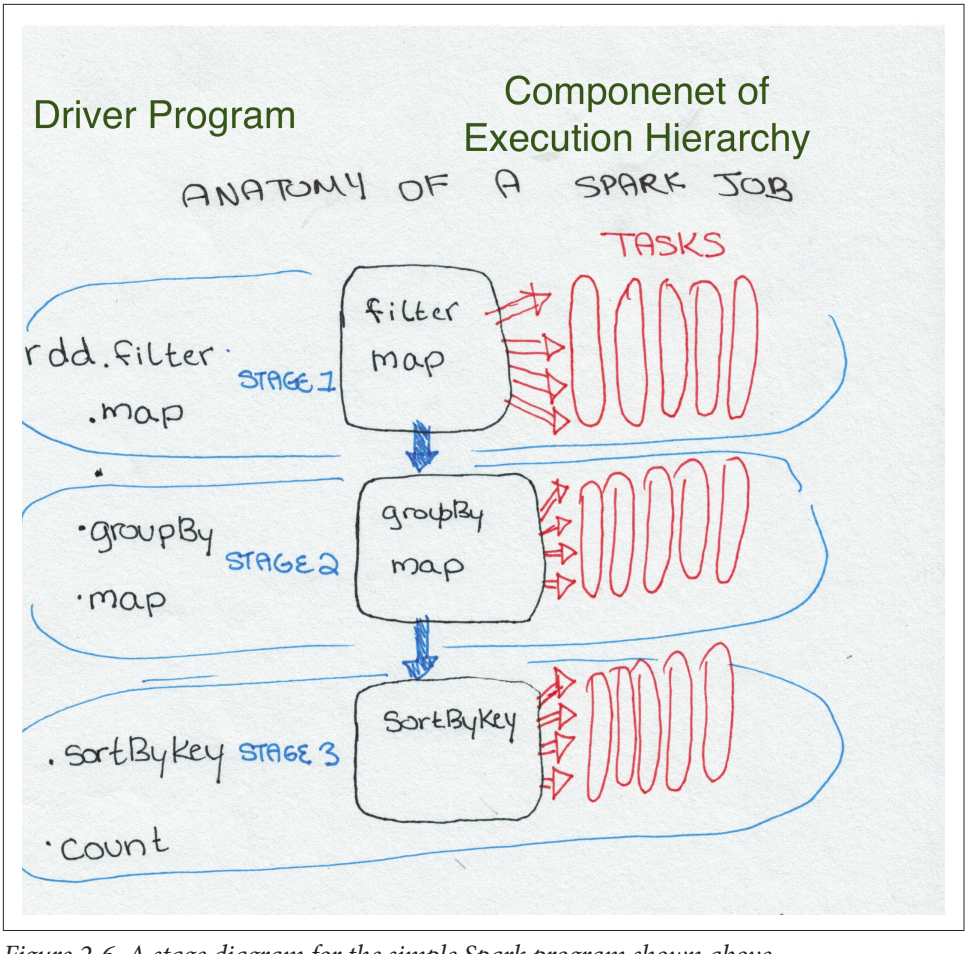


Figure 2-6. A stage diagram for the simple Spark program shown above.

A cluster cannot necessarily run every tasks in parallel for each stage. Each executor has a number of cores, configured at the application level, but likely corresponding to the physical cores on a cluster, for running tasks.. Spark can run no more tasks at once than the total number of executor cores allocated for the application. We can calculate this number from with the settings from the Spark Conf as (total number of executor cores = # of cores per executor * number of executors). If there are more partitions (and thus more tasks) then the number of slots for running tasks, the extra tasks will be allocated to the executors as the first round of tasks finish and resources are available. In most cases all the tasks for one stage must be completed before the next stage can start. The process of distributing these tasks is done by the `TaskScheduler` and varies depending on whether the fair scheduler discussed in ??? is used.

In some ways, the simplest way to think of the Spark execution model is that a Spark job is the set of RDD transformations needed to compute one final result. Each stage corresponds to a segment of work, which can be accomplished without involving the driver. In other words, one stage can be computed without moving data across the partitions. Within one stage, the tasks are the units of work done for each partition of the data.

Conclusion

Spark has an innovative, efficient model of parallel computing centering on lazily evaluated, immutable, distributed datasets, known as RDDs. Spark exposes RDDs as an interface, and RDD methods can be used without any knowledge of their implementation - but having an understanding of the details will help you write more performance code. Because of Spark's ability to run jobs concurrently, to compute jobs across multiple nodes, and to materialize RDDs lazily, the performance implications of similar logical patterns may differ widely and errors may surface from misleading places. Thus, it is important to understand how the execution model for your code is assembled in order to write and debug Spark code. Furthermore, it is often possible to accomplish the same tasks in many different ways using the Spark API, and a strong understanding of how your code is evaluated will help you optimize its performance. In this book, we will focus on ways to design Spark applications to minimize network traffic, memory errors, and the cost of failures.

CHAPTER 3

DataFrames, Datasets & Spark SQL

Spark SQL and its DataFrames and Datasets interfaces are the future of Spark performance, with more efficient storage options, advanced optimizer, and direct operations on serialized data. These components are super important for getting the best of Spark performance - see Figure 3-1.

Figure 3-1. Spark SQL Performance Relative Simple RDDs From SimplePerfTest.scala Aggregating avg Fuzziness

These are relatively new components; Datasets was introduced in Spark 1.6, DataFrames in Spark 1.3, and the SQL engine in Spark 1.0. This chapter is focused on

helping you learn how to best use Spark SQL's tools. For tuning params a good follow up is ???.

Spark's DataFrames have very different functionality compared to traditional DataFrames like Panda's and R. While these all deal with structured data, it is important not to depend on your existing intuition surrounding DataFrames.

Like RDDs, DataFrames and Datasets represent distributed collections, with additional schema information not found in RDDs. This additional schema information is used to provide a more efficient storage layer (Tungsten) and in the optimizer (Catalyst). Beyond schema information, the operations performed on DataFrames are such that the optimizer can inspect the logical meaning rather than arbitrary functions. Datasets are an extension of DataFrames bringing strong types, like with RDDs, for Scala/Java and more RDD-like functionality within the Spark SQL optimizer. Compared to working with RDDs, DataFrames allow Spark's optimizer to better understand our code and our data, which allows for a new class of optimizations we explore in "Query Optimizer" on page 77.

While Spark SQL, DataFrames, and Datasets provide many excellent enhancements, they still have some rough edges compared to traditional processing with "regular" RDDs. The Dataset API, being brand new at the time of this writing, is likely to experience some changes in future versions.

Getting Started with the HiveContext (or SQLContext)

Much as the `SparkContext` is the entry point for all Spark applications, and the `StreamingContext` is for all streaming applications, the `HiveContext` and `SQLContext` serve as the entry points for Spark SQL. The names of these entry points can be a bit confusing, and it is important to note the `HiveContext` **does not require** a Hive installation. The primary reason to use the `SQLContext` is if you have conflicts with the Hive dependencies that cannot be resolved. The `HiveContext` has a more complete SQL parser as well as additional user defined functions (UDFs) [1] and should be used whenever possible.

Like with all of the Spark components, you need to import a few extra components as shown in Example 3-1. If you are using the `HiveContext` you should import those

1 UDFs allow us to extend SQL to have additional powers, such as computing the geo-spatial distance between points

components as well. Once you have imported the basics, you can create the entry point as in Example 3-2.

Example 3-1. Spark SQL imports

```
import org.apache.spark.sql.{DataFrame, SparkSession, SQLContext, Row}
import org.apache.spark.sql.catalyst.expressions.aggregate._
import org.apache.spark.sql.expressions._
import org.apache.spark.sql.functions._
import org.apache.spark.sql.hive.HiveContext
import org.apache.spark.sql.hive.thriftserver._
```

Scala's type alias of `DataFrame = Dataset[Row]` is broken in Java - you must write `Dataset<Row>`.

Example 3-2. Creating the HiveContext

```
val hiveContext = new HiveContext(sc)
// Import the implicits, unlike in core Spark the implicits are defined on the context
import hiveContext.implicits._
```

To use the `HiveContext` you will need to add both Spark's `SQL` and `Hive` components to your dependencies. If you are using the sbt-spark-package plugin you can do this by just adding `SQL` and `Hive` to your list of `sparkComponents`.

Example 3-3. Add Spark SQL & Hive component to sbt-spark-package build

```
sparkComponents ++= Seq("sql", "hive", "hive-thriftserver", "hive-thriftserver")
```

For maven compatible build systems, the coordinates for Spark's SQL and Hive components in 2.0.0 are `org.apache.spark:spark-sql_2.11:2.0.0` and `org.apache.spark:spark-hive_2.11:2.0.0`.

Example 3-4. Add Spark SQL & Hive component to "regular" sbt build

```
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-sql" % "2.0.0",
  "org.apache.spark" %% "spark-hive" % "2.0.0")
```

Example 3-5. Add Spark SQL & Hive component to maven pom file

```
 <dependency> <!-- Spark dependency -->
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
```

```
  <version>2.0.0</version>
</dependency>
<dependency> <!-- Spark dependency -->
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-hive_2.11</artifactId>
  <version>2.0.0</version>
</dependency>
```

While it's not required, if you do have an existing Hive Metastore to which you wish to connect your `HiveContext`, you can copy your `hive-site.xml` to Spark's `conf/` directory.

As of Spark 1.6.0, the default Hive Metastore version is 1.2.1. For other versions of the Hive Metastore you will need to set the `spark.sql.hive.metastore.version` property to the desired versions as well as set `spark.sql.hive.metastore.jars` to either "maven" (to have Spark retrieve the jars) or the system path where the Hive JARs are present.

If you can't include the Hive dependencies with our application, you can leave out Spark's Hive component and instead create a `SQLContext`. This provides much of the same functionality, but uses a less capable SQL parser and lacks certain Hive based user defined functions (UDFs) and user defined aggregate functions (UDAFs).

Example 3-6. Creating the SQLContext

```
val sqlContext = new SQLContext(sc)
// Import the implicits, unlike in core Spark the implicits are defined on the context
import sqlContext.implicits._
```

If you are using the Spark Shell you will automatically get a SQLContext. This SQLContext will be a HiveContext if Spark was built with Hive support (e.g. `-PHive`).

As with the core Spark Context and Streaming Context, the Hive/SQLContext is used to load your data. JSON is a very popular format, in part because it can be easily loaded in many languages, and is at least semi-human-readable. Some of the sample data we've included in the book are in JSON format for exactly these reasons. JSON is especially interesting since it lacks schema information, and Spark needs to do some work to infer the schema from our data. JSON can also be expensive to parse, in some simple cases parsing the input JSON data can be greater than the actual operation. We will cover the full loading and saving API for JSON in "JSON" on page 61, but to get started let's load a sample we can use to explore the schema.

Example 3-7. Load JSON sample

```
val df1 = sqlCtx.read.json(path)
```

Now that we've got the JSON data loaded we start by exploring what schema Spark has managed to infer for our data.

Basics of Schemas

The schema information, and the optimizations it enables, is one of the core differences between Spark SQL and core Spark. Inspecting the schema is especially useful for DataFrames since you don't have the templated type you do with RDDs or Datasets. Schemas are normally handled automatically by Spark SQL, either inferred when loading the data or computed based on the parent DataFrames and the transformation being applied.

DataFrames expose the schema in both human-readable or programmatic formats. `printSchema()` will show us the schema of a DataFrame and is most commonly used when working in the shell to figure out with what you are working. This is especially useful for data formats, like JSON, where the schema may not be immediately visible by looking at only a few records or reading a header. For programmatic usage, you can get the schema by simply calling `schema`, which is often used in ML pipeline transformers. Since you are likely familiar with case classes and JSON, let's examine how the equivalent Spark SQL schema would be represented.

Example 3-8. JSON data which would result in an equivalent schema

```
{"name":"mission","pandas":[{"id":1,"zip":"94110","pt":"giant", "happy":true,
                             "attributes":[0.4,0.5]}]}
```

Example 3-9. Equivalent case class to the next two examples

```
case class RawPanda(id: Long, zip: String, pt: String, happy: Boolean, attributes: Array[Double])
case class PandaPlace(name: String, pandas: Array[RawPanda])
```

Example 3-10. StructField case class

```
case class StructField(
    name: String,
    dataType: DataType,
    nullable: Boolean = true,
    metadata: Metadata = Metadata.empty)
....
```

Example 3-11. Sample schema information for nested structure (`.schema()`) - manually formatted

```
org.apache.spark.sql.types.StructType = StructType(
  StructField(name,StringType,true),
  StructField(pandas,
    ArrayType(
      StructType(StructField(id,LongType,false),
                    StructField(zip,StringType,true),
                    StructField(pt,StringType,true),
                    StructField(happy,BooleanType,false),
                    StructField(attributes,ArrayType(DoubleType,false),true)),true),true))
```

Example 3-12. Sample Schema Information for Nested Structure (`.printSchema()`)

```
root
 |-- name: string (nullable = true)
 |-- pandas: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- id: long (nullable = false)
 |    |    |-- zip: string (nullable = true)
 |    |    |-- pt: string (nullable = true)
 |    |    |-- happy: boolean (nullable = false)
 |    |    |-- attributes: array (nullable = true)
 |    |    |    |-- element: double (containsNull = false)
```

From here we can dive into what this schema information means and look at how to construct more complex schemas. The first part is a `StructType` which contains a list of fields. It's important to note you can nest `StructTypes`, like how a case class can contain additional case classes. The fields in the `StructType` are defined with `StructField` which specifies the name, type (see Table 3-1 and Table 3-2 for a listing of types), and a Boolean indicating if the field may be null/missing.

Table 3-1. Basic Spark SQL types

Scala Type	SQL Type	Details
Byte	ByteType	1 byte signed integers (-128,127)
Short	ShortType	2 byte signed integers (-32768,32767)
Int	IntegerType	4 byte signed integers (-2147483648,2147483647)
Long	LongType	8 byte signed integers (-9223372036854775808,9223372036854775807)
java.math.BigDecimal	DecimalType	Arbitrary precision signed decimals
Float	FloatType	4 byte floating point number

Scala Type	SQL Type	Details
Double	DoubleType	8 byte floating point number
Array[Byte]	BinaryTpe	Array of bytes
Boolean	BooleanTpe	true/false
java.sql.Date	DateType	Date without time information
java.sql.Timestamp	TimestampType	Date with time information (second precision)
String	StringType	Character string values (stored as UTF8)

Table 3-2. Complex Spark SQL types

Scala Type	SQL Type	Details	Example
Array[T]	ArrayType(elementType, containsNull)	Array of single type of element, containsNull true if any null elements.	`Array[Int] => Array Type(IntegerType, true)`
Map[K, V]	MapType(elementType, valueType, valueContainsNull)	Key/value map, valueContainsNull if any values are null.	`Map[String, Int] => Map Type(StringType, Integer Type, true)`
case class	StructType(`List[Struct Fields]`)	Named fields of possible heterogeneous types, similar to a case class or JavaBean.	`case class Panda(name: String, age: Int) => StructType(List(Struct Field("name", StringType, true), StructField("age", IntegerType, true)))`

As you saw in Example 3-11, you can nest StructFields and all of the complex Spark SQL types.

Now that you've got an idea of how to understand and, if needed, specify schemas for your data, you are ready to start exploring the DataFrame interfaces.

Spark SQL schemas are eagerly evaluated, unlike the data underneath. If you find yourself in the shell and uncertain of what a transformation will do - try it and print the schema. See Example 3-12.

DataFrame API

Spark SQL's DataFrame API allows us to work with DataFrames without having to register temporary tables or generate SQL expressions. The DataFrame API has both transformations and actions. The transformations on DataFrames are more relational in nature, with the Dataset API (covered next) offering a more functional-style API.

Transformations

Transformations on DataFrames are similar in concept to RDD transformations, but with a more relational flavor. Instead of specifying arbitrary functions, which the optimizer is unable to introspect, you use a restricted expression syntax so the optimizer can have more information. As with RDDs, we can broadly break down transformations into simple single DataFrame, multiple DataFrame, key/value, and grouped/windowed transformations.

Spark SQL transformations are only partially lazy; the schema is eagerly evaluated.

Simple DataFrame Transformations & SQL Expressions

Simple DataFrame transformations allow us to do most of the standard things one can do when working a row at a time.[1] You can still do many of the same operations defined on RDDs, except using Spark SQL expressions instead of arbitrary functions. To illustrate this we will start by examining the different kinds of filter operations available on DataFrames.

DataFrame functions, like `filter` accept Spark SQL expressions instead of lambdas. These expressions allow the optimizer to understand what the condition represents, and with filter, it can often be used to skip reading unnecessary records.

To get started, let's look at a SQL expression to filter our data for unhappy pandas using our existing schema. The first step is looking up the column that contains this information. In our case it is "happy", and for our DataFrame (called df) we access

1 A row at a time allows for narrow transformations with no shuffle.

the column through the apply function (e.g. (`df("happy")`)). The `filter` expression requires the expression to return a boolean value, and if you wanted to select happy pandas, the entire expression could be retrieving the column value. However, since we want to find the unhappy pandas though, we can check to see that happy isn't true using the `!==` operator as shown in Example 3-13.

Example 3-13. Simple filter for unhappy pandas

```
pandaInfo.filter(pandaInfo("happy") !== true)
```

To lookup op the column, we can either provide the column name on the specific DataFrame or use the implicit `$` operator for column lookup. This is especially useful when the DataFrame is anonymous. The `!` binary negation function can be used together with `$` to simplify our expression from Example 3-13 down to `df.filter(!$("happy"))`.

This illustrates how to access a specific column from a DataFrame. For accessing other structures inside of DataFrames, like nested structs, keyed maps, and array elements, use the same apply syntax. So, if the first element in the attributes array represent squishiness, and you only want very squishy pandas, you can access that element by writing `df("attributes")(0) >= 0.5`.

Our expressions need not be limited to a single column. You can compare multiple columns in our "filter" expression. Complex filters like this are more difficult to push down to the storage layer, so you may not see the same speedup over RDDs that you see with simpler filters.

Example 3-14. More complex filter

```
pandaInfo.filter(
  pandaInfo("happy").and(pandaInfo("attributes")(0) > pandaInfo("attributes")(1))
)
```

Spark SQL's column operators are defined on the column class, so a filter containing the expression `0 >= df.col("friends")` will not compile since Scala will use the `>=` defined on 0. Instead you would write `df.col("friend") <= 0` or convert 0 to a column literal with `lit` [1].

1 A column literal is a column with a fixed value that doesn't change between rows (i.e. constant).

Spark SQLs DataFrame API has a very large set of operators available. You can use all of the standard mathematical operators on floating points, along with the standard logical and bitwise operations (prefix with `bitwise` to distinguish from logical). Columns use `===` and `!==` for equality to avoid conflict with Scala internals. For columns of strings, `startsWith`/`endsWith`, `substr`, `like`, and `isNull` are all available. The full set of operations is listed in org.apache.spark.sql.Column and covered in Table 3-3 and Table 3-4.

Table 3-3. Spark SQL Scala operators

Scala Operator	Java Equivalent	Input Column Types	Output Type	Purpose	Sample	Result
`!==`	`notEqual`	Any	Boolean	Check if expressions not equal	"hi" !== "bye"	true
`%`	`mod`	Numeric	Numeric	Modulo	`10 % 5`	0
`&&`	`and`	Boolean	Boolean	Boolean and	`true && false`	`false`
`*`	`multiply`	Numeric	Numeric	Multiply expressions	`2 * 21`	42
`+`	`plus`	Numeric	Numeric	Sum expression	`2 + 2`	4
`-`	`minus`	Numeric	Numeric	Subtraction	`2 - 2`	0
`-`	`unary_-`	Numeric	Numeric	Unary subtraction	`-42`	`-42`
`/`	`division`	Numeric	Double	Division	`43/2`	`21.5`
`<`	`lt`	Comparable	Boolean	Less than	`"a" < "b"`	`true`
`<=`	`leq`	Comparable	Boolean	Less than or equal to	`"a" <= "a"`	`true`
`===`	`equals`	Any	Any	Equality test (unsafe on null values)	`"a" === "a"`	`true`
`<=>`	`eqNullSafe`	Any	Any	Equality test (safe on null values)	`"a" <=> "a"`	`true`
`>`	`gt`	Comparable	Boolean	Greater than	`"a" > "b"`	`false`
`>=`	`gt`	Comparable	Boolean	Greater than or equal to	`"a" >= "b"`	`false`

Table 3-4. Spark SQL expression operators

Operator	Input Column Types	Output Type	Purpose	Sample	Result
`apply`	Complex types	Type of field accessed	Get value from complex type (e.g. structfield/map lookup or array index)	`[1,2,3].apply(0)`	`1`
`bitwiseAND`	Integral Type [a]	Same as input	Computes and bitwise	`21.bitwiseAND(11)`	`1`
`bitwiseOR`	Integral Type [a]	Same as input	Computes or bitwise	`21.bitwiseOR(11)`	`31`
`bitwiseXOR`	Integral Type [a]	Same as input	Computes bitwise exclusive or	`21.bitwiseXOR(11)`	`30`

[a] Integral types include ByteType, IntegerType, LongType, and ShortType.

Not all Spark SQL expressions can be used in every API call. For example, Spark SQL joins do not support complex operations, and filter requires that the expression result in a boolean, and similar.

In addition to the operators directly specified on the column, an even larger set of functions on columns exists in org.apache.spark.sql.functions, some of which we cover in tables Table 3-5 and Table 3-6. For illustration, this example shows the values for each column at a specific row, but keep in mind that, these functions are called on columns not values.

Table 3-5. Spark SQL standard functions

Function name	Purpose	Input Types	Example usage	Result
`lit(value)`	Convert a Scala symbol to a column literal [3]	Column & Symbol	`lit(1)`	`Column(1)`
`array`	Create a new array column	Must all have the same Spark SQL type	`array(lit(1),lit(2))`	array(1,2)
`isNaN`	Check if not a number	Numeric	`isNan(lit(100.0))`	false
`not`	Opposite value	Boolean	`not(lit(true))`	false

Table 3-6. Spark SQL Common Mathematical Expressions

Function name	Purpose	Input Types	Example usage	Result
`abs`	Absolute value	Numeric	`abs(lit(-1))`	1
`sqrt`	Square root	Numeric	`sqrt(lit(4))`	2
`acos`	Inverse cosine	Numeric	`acos(lit(0.5))`	1.04….[a]
`asin`	Inverse sine	Numeric	`asin(lit(0.5))`	0.523…[a]
`atan`	Inverse tangent	Numeric	`atan(lit(0.5))`	0.46…[a]
`cbrt`	Cube root	Numeric	`sqrt(lit(8))`	2
`ceil`	Ceiling	Numeric	`ceil(lit(8.5))`	9
`cos`	Cosine	Numeric	`cos(lit(0.5))`	0.877….[a]
`sin`	Sine	Numeric	`sin(lit(0.5))`	0.479…[a]
`tan`	Tangent	Numeric	`tan(lit(0.5))`	0.546…[a]
`exp`	Exponent	Numeric	`exp(lit(1.0))`	2.718…[a]
`floor`	Ceiling	Numeric	`floor(lit(8.5))`	8
`least`	Minimum value	Numerics	`least(lit(1), lit(-10))`	-10

[a] Truncated for display purposes.

Table 3-7. Functions for use on Spark SQL arrays

Function name	Purpose	Example usage	Result
`array_contains`	If an array contains a value	`array_contains(lit(Array(2,3,-1), 3))`	true
`sort_array`	Sort an array (ascending default)	`sort_array(lit(Array(2,3,-1)))`	`Array(-1,2,3)`

Function name	Purpose	Example usage	Result
`explode`	Create a row for each element in the array - often useful when working with nested JSON records. Either takes a column name or additional function mapping from row to iterator of case classes.	`explode(lit(Array(2,3,-1)), "murh")`	`Row(2)`, `Row(3)`, `Row(-1)`

Beyond simply filtering out data, you can also produce a DataFrame with new columns or updated values in old columns. Spark uses the same expression syntax we discussed for filter, except instead of having to include a condition (like testing for equality), the results are used as values in the new DataFrame. To see how you can use `select` on complex and regular data types, Example 3-15 uses the Spark SQL `explode` function to turn an input DataFrame of `PandaPlaces` into a DataFrame of just `PandaInfo` as well as computing the "squishness" to "hardness" ratio of each panda.

Example 3-15. Spark SQL select and explode operators

```
val pandaInfo = pandaPlace.explode(pandaPlace("pandas")){
  case Row(pandas: Seq[Row]) =>
    pandas.map{
      case Row(id: Long, zip: String, pt: String, happy: Boolean, attrs: Seq[Double]) =>
        RawPanda(id, zip, pt, happy, attrs.toArray)
    }}
pandaInfo.select(
  (pandaInfo("attributes")(0) / pandaInfo("attributes")(1))
    .as("squishyness"))
```

When you construct a sequence of operations, the generated column names can quickly become unwieldy, so the `as` or `alias` operator are useful to specify the resulting column name.

While all of these operations are quite powerful, sometimes the logic you wish to express is more easily encoded with `if/else` semantics. A simple example of this is encoding the different types of panda as a numeric value [1]. The `when` and `otherwise` functions can be chained together to create the same effect.

1 StringIndexer in the ML pipeline is designed for string index encoding.

Example 3-16. If/Else in Spark SQL

```
/**
  * Encodes pandaType to Integer values instead of String values.
  *
  * @param pandaInfo the input DataFrame
  * @return Returns a DataFrame of pandaId and integer value for pandaType.
  */
def encodePandaType(pandaInfo: DataFrame): DataFrame = {
  pandaInfo.select(pandaInfo("id"),
    (when(pandaInfo("pt") === "giant", 0).
    when(pandaInfo("pt") === "red", 1).
    otherwise(2)).as("encodedType")
  )
}
```

Specialized DataFrame Transformations for Missing & Noisy Data

Spark SQL also provides special tools for handling missing, null, and invalid data. By using `isNan` or `isNull` along with filters, you can create conditions for the rows you want to keep. For example, if you have a number of different columns, perhaps with different levels of precision (some of which may be null), you can use `coalesce(c1, c2, ..)` to return the first non-null column. Similarly, for numeric data, `nanvl` returns the first non-NaN value (e.g. nanvl(0/0, sqrt(-2), 3) results in 3). To simplify working with missing data, the `na` function on DataFrame gives us access to some common routines for handling missing data in DataFrameNaFunctions.

Beyond Row-by-Row Transformations

Sometimes applying a row-by-row decision, as you can with `filter`, isn't enough. Spark SQL also allows us to select the unique rows by calling `dropDuplicates`, but as with the similar operation on RDDs (`distinct`), this can require a shuffle, so is often much slower than filter. Unlike with RDDs, `dropDuplicates` can optionally drop rows based on only a subset of the columns, such as an id field.

Example 3-17. Drop duplicate panda ids

```
pandas.dropDuplicates(List("id"))
```

This leads nicely into our next section on aggregates and groupBy since often the most expensive component of each is the shuffle.

Aggregates and `groupBy`

Spark SQL has many powerful aggregates, and thanks to its optimizer it can be easy to combine many aggregates into one single action/query. Like with Pandas' DataFrames, `groupBy` returns a special `GroupedData` object on which we can ask for cer-

tain aggregations to be performed. Aggregations on Datasets behave differently returning a `GroupedDataset`, as discussed in "Grouped Operations on Datasets" on page 74. `min`, `max`, `avg`, and `sum` are all implemented as convenience functions directly on `GroupedData`, and more can be specified by providing the expressions to `agg`. Once you specify the aggregates you want to compute, you can get the results back as a DataFrame.

If you're used to RDDs you might be concerned by `groupBy`, but it is now a safe operation on DataFrames thanks to the Spark SQL optimizer, which automatically pipelines our reductions avoiding giant shuffles and mega records.

Example 3-18. Compute the max panda size by zip code

```
def maxPandaSizePerZip(pandas: DataFrame): DataFrame = {
  pandas.groupBy(pandas("zip")).max("pandaSize")
}
```

While this computes the max on a per-key basis, these aggregates can also be applied over the entire Dataframe or all numeric columns in a Dataframe. This is often useful when trying to collect some summary statistics for the data with which you are working. In fact there is a built-in `describe` transformation which does just that, although it can also be limited to certain columns.

Example 3-19. Compute some common summary stats, including count, mean, stddev, and more, on the entire DataFrame

```
pandas.describe()
```

The behaviour of groupBy has changed between Spark versions. Prior to Spark 1.3 the values of the grouping columns are discarded by default, while post 1.3 they are retained. The configuration parameter, `spark.sql.retainGroupColumns`, can be set to false to force the earlier functionality.

For computing multiple different aggregations, or more complex aggregations, you should use the `agg` api on the `GroupedData` instead directly calling `count`, `mean`, or similar convenience functions. For the `agg` API, you either supply a list of aggregate expressions, a string representing the aggregates, or a map of column names to aggregate function name. Once we've called `agg` with the requested aggregates, we get back a regular DataFrame with the aggregated results. As with regular functions, they are listed in org.apache.spark.sql.functions scaladoc. Table 3-8 lists some common and useful aggregates. For our example results in these tables we will consider a Data-

Frame with the schema of name field (as a string) and age (as an integer), both nullable with values ({"ikea", null}, {"tube", 6}, {"real", 30}).

Example 3-20. Example aggregates using the agg API

```
def minMeanSizePerZip(pandas: DataFrame): DataFrame = {
  // Compute the min and mean
  pandas.groupBy(pandas("zip")).agg(min(pandas("pandaSize")), mean(pandas("pandaSize")))
}
```

Computing multiple aggregates with Spark SQL can be much simpler than doing the same tasks with the RDD API.

Table 3-8. Spark SQL aggregate functions for use with agg API

Function name	Purpose	Storage requirement	Input Types	Example usage	Example Result
`approxCountDistinct`	Count approximate distinct values in column [a]	Configurable through rsd (which controls error rate)	All	`df.agg(approxCountDistinct(df("age"), 0.001))`	2
`avg`	Average	Constant	Numeric	`df.agg(avg(df("age")))`	18
`count`	Count number of items (excluding nulls). Special case of "*" counts number of rows	Constant	All	`df.agg(count(df("age")))`	2
`countDistinct`	Count distinct values in column	O(distinct elems)	All	`df.agg(countDistinct(df("age")))`	2
`first`	Return the first element [b]	Constant	All	`df.agg(first(df("age")))`	6
`last`	Return the last element	Constant	All	`df.agg(last(df("age")))`	30

Function name	Purpose	Storage requirement	Input Types	Example usage	Example Result
`stddev`	Sample standard deviation [c]	Constant	Numeric	`df.agg(stddev(df("age")))`	16.97…
`stddev_pop`	Population standard deviation [c]	Constant	Numeric	`df.agg(stddev_pop(df("age")))`	12.0
`sum`	Sum of the values	Constant	Numeric	`df.agg(sum(df("age")))`	36
`sumDistinct`	Sum of the distinct values	0(distinct elems)	Numeric	`df.agg(sumDistinct(df("age")))`	36
`min`	Select the minimum value	Constant	Sortable data	`df.agg(min(df("age")))`	5
`max`	Select the average value	Constant	Sortable data	`df.agg(max(df("age")))`	30
`mean`	Select the maximum value	Constant	Sortable data	`df.agg(max(df("age")))`	30

[a] Implemented with HyperLogLog: *https://en.wikipedia.org/wiki/HyperLogLog* .

[b] This was commonly used in early versions of Spark SQL where the grouping column was not preserved.

[c] Added in Spark 1.6.

In addition aggregates on `groupBy`, you can run the same aggregations on multi-dimensional cubes with `cube` and rollups with `rollup`.

If the built-in aggregation functions don't meet your needs, you can extend Spark SQL using UDFS as discussed in "Extending with User Defined Functions & Aggregate Functions (UDFs, UDAFs)" on page 74, although things can be more complicated for aggregate functions.

Windowing

Spark SQL 1.4.0 introduced windowing functions to allow us to more easily work with ranges or windows of rows. When creating a window you specify what columns the window is over, the order of the rows within each partition/group, and the size of the window (e.g. K rows before and J rows after OR range between values). Using this specification each input row is related to some set of rows, called a frame, that is used to compute the resulting aggregate. Window functions can be very useful for things like computing average speed with noisey data, relative sales, and more.

Example 3-21. Define a window on the +/-10 closest (by age) pandas in the same zip code

```
val windowSpec = Window
  .orderBy(pandas("age"))
  .partitionBy(pandas("zip"))
  .rowsBetween(start = -10, end = 10) // can use rangeBetween for range instead
```

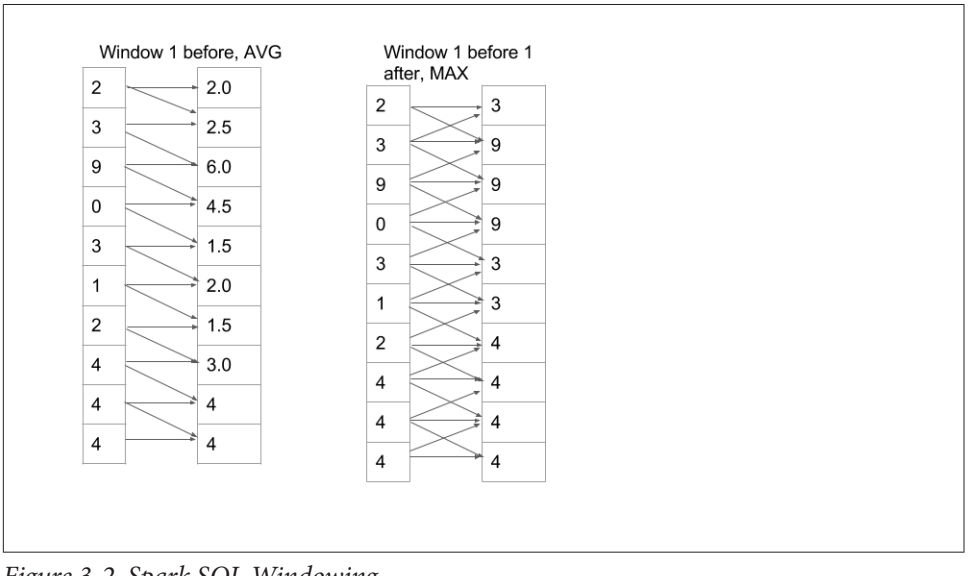


Figure 3-2. Spark SQL Windowing

Once you've defined a window specification you can compute a function over it. Spark's existing aggregate functions, covered in "Aggregates and groupBy" on page 52, can be computed an aggregate for the window. Window operations are very useful for things like Kalman filtering or many types of relative analysis.

Example 3-22. Compute difference from the average using the window of +/-10 closest (by age) pandas in the same zip code

```
val pandaRelativeSizeCol = pandas("pandaSize") -
  avg(pandas("pandaSize")).over(windowSpec)

pandas.select(pandas("name"), pandas("zip"), pandas("pandaSize"), pandas("age"),
  pandaRelativeSizeCol.as("panda_relative_size"))
```

As of this writing, Windowing functions require a HiveContext.

Sorting

Sorting supports multiple columns in ascending or descending order, with ascending as the default. Spark SQL has some extra benefits for sorting as some serialized data can be compared without deserialization.

Example 3-23. Sort by panda age and size in opposite orders

```
pandas.orderBy(pandas("pandaSize").asc, pandas("age").desc)
```

When limiting results, sorting is often used to only bring back the top or bottom K results. When limiting you specify the number of rows with `limit(numRows)` to restrict the number of rows in the DataFrame. Limits are also sometimes used for debuging without sorting to bring back a small result. If, instead of limiting the number of rows based on a sort order, you want to sample your data, ??? covers techniques for Spark SQL sampling as well.

Multi DataFrame Transformations

Beyond single DataFrame transformations you can perform operations that depend on multiple DataFrames. The ones that first pop into our heads are most likely the different types of joins, which are covered in Chapter 4, but beyond that you can also perform a number of set-like operations between DataFrames.

Set Like Operations

The DataFrame set-like operations allow us to perform many operations that are most commonly thought of as set operations. These operations behave a bit differently than traditional set operations since we don't have the restriction of unique elements. While you are likely already familiar with the results of set-like operations from regular Spark and Learning Spark, it's important to review the cost of these operations in Table 3-9.

Table 3-9. Set operations

Operation Name	Cost
`unionAll`	Low
`intersect`	Expensive
`except`	Expensive
`distinct`	Expensive

Plain Old SQL Queries and Interacting with Hive Data

Sometimes, it's to use regular SQL queries instead of building up our operations on DataFrames. If you are connected to a Hive Metastore we can directly write SQL queries against the Hive tables and get the results as a `DataFrame`. If you have a DataFrame you want to write SQL queries against, you can register it as a temporary table (or save it as a managed table if you intend to re-use it between jobs). Datasets can also be converted back to DataFrames and registered for querying against.

Example 3-24. Registering/saving tables

```
def registerTable(df: DataFrame): Unit = {
  df.registerTempTable("pandas")
  df.write.saveAsTable("perm_pandas")
}
```

Querying tables is the same, regardless of whether it is a temporary table, existing Hive table, or newly-saved Spark table.

Example 3-25. Querying a table (permanent or temporary)

```
def querySQL(): DataFrame = {
  sqlContext.sql("SELECT * FROM pandas WHERE size > 0")
}
```

In addition to registering tables you can also write queries directly against a specific file path.

Example 3-26. Querying a raw file

```
def queryRawFile(): DataFrame = {
  sqlContext.sql("SELECT * FROM parquet.`path_to_parquet_file`")
}
```

Data Representation in DataFrames & Datasets

DataFrames are more than RDDs of `Row` objects; DataFrames and Datasets have a specialized representation and columnar cache format. The specialized representation is not only more space efficient, but also can be much faster to encode than even Kyro serialization. To be clear, like RDDs, DataFrames and Datasets are generally lazily evaluated and build up a lineage of their dependencies (except in DataFrames this is called a logical plan and contains more information).

Tungsten

Tungsten is a new Spark SQL component that provides more efficient Spark operations by working directly at the byte level. Looking back on, Figure 3-1, we can take a closer look at the space differences between the RDDs and DataFrames when cached in Figure 3-3. Tungsten includes specialized in-memory data structures tuned for the type of operations required by Spark, improved code generation, and a specialized wire protocol.

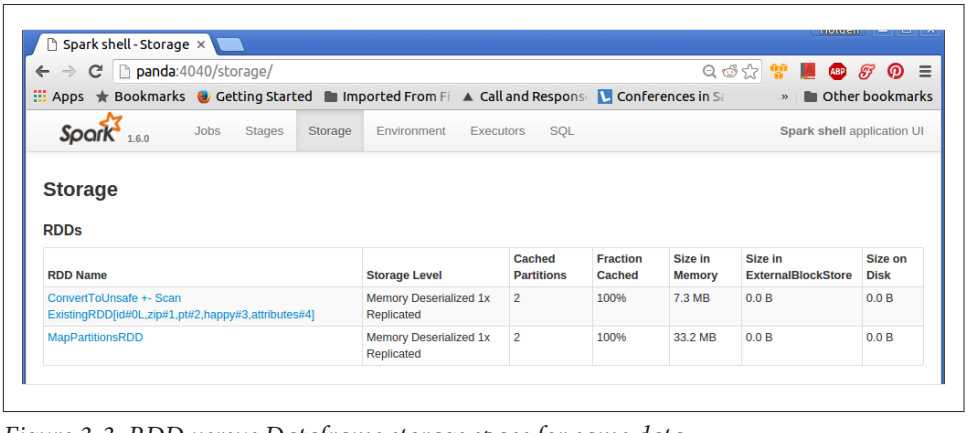


Figure 3-3. RDD versus Dataframe storage space for same data

For those coming from Hadoop, you can think of Tungsten data types as being `WritableComparable` types on steroids.

Tungsten's representation is substantially smaller than objects serialized using Java or even Kryo serializers. As Tungsten does not depend on Java objects, both on-heap and off-heap allocations are supported. Not only is the format more compact, serialization times can be substantially faster than with native serialization.

Since Tungsten no longer depends on working with Java objects, you can use either on-heap (in the JVM) or off-heap storage. If you use off-heap storage, it is important to leave enough room in your containers for the off-heap allocations - which you can get an approximate idea for from the web ui.

Tungsten's data structures are also created closely in mind with the kind of processing for which they are used. The classic example of this is with sorting, a common and expensive operation. The on-wire representation is implemented so that sorting can be done without having to deserialize the data again.

In the future Tungsten may make it more feasible to use certain non-JVM libraries. For many simple operations the cost of using BLAS, or similar linear algebra packages, from the JVM is dominated by the cost of copying the data off-heap.

By avoiding the memory and GC overhead of regular Java objects, Tungsten is able to process larger data sets than the same hand-written aggregations. Tungsten became the default in Spark 1.5 and can be enabled in earlier versions by setting `spark.sql.tungsten.enabled` to true (or disabled in later versions by setting this to false). Even without Tungsten, Spark SQL uses a columnar storage format with Kryo serialization to minimize storage cost.

Data Loading and Saving Functions

Spark SQL has a different way of loading and saving data than core Spark. So as to be able to push down certain types of operations to the storage layer, Spark SQL has its own Data Source API. Data Sources are able to specify and control which type of operations should be pushed down to the data source. As developers, you don't need to worry too much about the internal activity going on here, unless the data sources you are looking for are not supported.

Data loading in Spark SQL is not quite as lazy as in regular Spark, but is still generally lazy. You can verify this by quickly trying to load from a data source that doesn't exist.

DataFrameWriter and DataFrameReader

The DataFrameWriter and the DataFrameReader cover writing and reading from external data sources. The `DataFrameWriter` is accessed by calling `write` on a `Data`

Frame or Dataset. The DataFrameReader can be accessed through read on a SQLContext.

Spark SQL updated the load/save API in Spark 1.4, so you may see code still using the old-style API without the DataFrame reader or writer classes, but under the hood it is implemented as a wrapper around the new API.

Formats

When reading or writing you specify the format by calling format(formatName) on the DataFrameWriter/DataFrameReader. Format specific parameters, such as number of records to be sampled for JSON, are specified by either providing a map of options with options or setting option-by-option with option on the reader/writer.

The first party formats json, jdbc, orc, and parquet methods directly defined on the reader/writers taking the path or connection info. These methods are for convenience only and are wrappers around the more general methods we illustrate in this chapter.

JSON

Loading and writing JSON is supported directly in Spark SQL, and despite the lack of schema information in JSON, Spark SQL is able to infer a schema for us by sampling the records. Loading JSON data is more expensive than loading many data sources, since Spark needs to read some of the records to determine the schema information. If the schema between records varies widely (or the number of records is very small), you can increase the percentage of records read to determine the schema by setting samplingRatio to a higher value.

Example 3-27. Load JSON data, using all (100%) of records to determine the schema

```
val df2 = sqlCtx.read.format("json").option("samplingRatio", "1.0").load(path)
```

Spark's schema inference can be a compelling reason to use Spark for processing JSON data, even if the data size could be handled on a single node.

Since our input may contain some invalid JSON records you may wish to filter out, you can also take in an RDD of strings. This allows us to load the input as a standard text file, filter out our invalid records, and then load the data into JSON. This is done

by using the built-in `json` function on the DataFrameReader, which takes RDDs or paths. Methods for converting RDDs of regular objects are covered in "RDDs" on page 65.

Example 3-28. jsonRDD load

```
val rdd: RDD[String] = input.filter(_.contains("panda"))
val df = sqlCtx.read.json(rdd)
```

JDBC

The JDBC data source represent a natural Spark SQL data source, one which supports many of the same operations. Since different database vendors have slightly different JDBC implementations, you need to add the JAR for your JDBC data sources. Since SQL field types vary as well, Spark uses `JdbcDialects` with built-in dialects for DB2, Derby, MsSQL, Mysql, Oracle, and Postgres.[1]

While Spark supports many different JDBC sources, it does not ship with the JARs required to talk to all of these databases. If you are submitting your Spark job with `spark-submit` you can download the required JARs to the host you are launching and include them by specifying `--jars` or supply the maven coordinates to `--packages`. Since the Spark shell is also launched this way, the same syntax works and you can use it to include the MySQL JDBC JAR in Example 3-29.

Example 3-29. Include MySQL JDBC JAR

In earlier versions of Spark `--jars` does not include the JAR in the driver's class path. If this is the case for your cluster you must also specify the same JAR to `--driver-class-path`.

JdbcDialects allow Spark to correctly map the JDBC types to the corresponding Spark SQL types. If there isn't a JdbcDialect for your database vendor, the default dialect will be used which will likely work for many of the types. The dialect is automatically chosen based on the JDBC URL used.

If you find yourself needing to customize the JdbcDialect for your database vendor, you can look for a package or spark-packages or extend the `JdbcDialect` class and register your own dialect.

1 Some types may not be correctly implemented for all databases.

As with the other built-in data sources, there exists a convenience wrapper for specifying the properties required to load JDBC data. The convenience wrapper JDBC accepts the URL, table, and a `java.util.Properties` object for connection properties (such as authentication information). The properties object is merged with the properties that are set on the reader/writer itself. While the properties object is required, an empty properties object can be provided and properties instead specified on the reader/writer.

Example 3-30. Create a DataFrame from a JDBC data source

```
sqlContext.read.jdbc("jdbc:dialect:serverName;user=user;password=pass",
  "table", new Properties)

sqlContext.read.format("jdbc")
  .option("url", "jdbc:dialect:serverName")
  .option("dbtable", "table").load()
```

The API for saving a DataFrame is very similar to the API used for loading. The `save()` function needs no path since the information is already specified, just as with loading.

Example 3-31. Write a DataFrame to a JDBC data source

```
df.write.jdbc("jdbc:dialect:serverName;user=user;password=pass",
  "table", new Properties)

df.write.format("jdbc")
  .option("url", "jdbc:dialect:serverName")
  .option("user", "user")
  .option("password", "pass")
  .option("dbtable", "table").save()
```

In addition to reading and writing JDBC data sources, Spark SQL can also run its own JDBC server, which is covered later on in this chapter.

Parquet

Apache Parquet files are a common format directly supported in Spark SQL, and they are incredibly space efficient and popular. Apache Parquet's popularity comes from a number of features, including the ability to easily split across multiple files, compression, nested types, and many others discussed in the Parquet documentation. Since Parquet is such a popular format, there are some additional options available in Spark for the reading and writing of Parquet files. Unlike third party data sources, these options are mostly configured on the SQLContext, although some can be configured on either the `SQLContext` or `DataFrameReader/Writer`.

Table 3-10. Parquet Data Source Options

SQLConf	DataFrameReader/ Writer option	Default	Purpose
spark.sql.parquet.mergeSchema	mergeSchema	False	Control if schema should be merged between partitions when reading. Can be expensive, so disabled by default in 1.5.0.
spark.sql.parquet.binaryAsString	N/A	False	Treat binary data as strings. Old versions of Spark wrote strings as binary data.
spark.sql.parquet.cacheMetadata	N/A	True	Cache parquet metadata, normally safe unless underlying data is being modified by another process.
spark.sql.parquet.compression.codec	N/A	Gzip	Specify the compression codec for use with parquet data. Valid options are uncompressed, snappy, gzip, or lzo.
spark.sql.parquet.filterPushdown	N/A	True	Push down filters to parquet (when possible) [a].
spark.sql.parquet.writeLegacyFormat	N/A	False	Write in parquet metadata in the legacy format.
spark.sql.parquet.output.committer.class	N/A	`org.apache.parquet.hadoop.ParquetOutputCommitter`	Output committer used by parquet. If writing to S3 you may wish to try `org.apache.spark.sql.parquet.DirectParquetOutputCommitter`.

[a] Pushdown means evaluate at the storage, so with parquet this can often mean skipping reading unnecessary rows or files.

Example 3-32. Read Parquet file written by an old version of Spark

```
def loadParquet(path: String): DataFrame = {
  // Configure Spark to read binary data as string, note: must be configured on SQLContext
  sqlContext.setConf("spark.sql.parquet.binaryAsString", "true")

  // Load parquet data using merge schema (configured through option)
  sqlContext.read
    .option("mergeSchema", "true")
    .format("parquet")
    .load(path)
}
```

Example 3-33. Write Parquet file with default options

```
def writeParquet(df: DataFrame, path: String) = {
  df.write.format("parquet").save(path)
}
```

Hive Tables

Interacting with Hive tables adds another option beyond the other formats. As covered in "Plain Old SQL Queries and Interacting with Hive Data" on page 58, one option for bringing in data from a Hive table is writing a SQL query against it and having the result as a DataFrame. The DataFrame's reader and writer interfaces can also be used with Hive tables as with the rest of the data sources.

Example 3-34. Load a Hive table

```
def loadHiveTable(): DataFrame = {
  sqlContext.read.table("pandas")
}
```

When loading a Hive table Spark SQL will convert the metadata and cache the result, if the underlying metadata has changed you can use `sqlContext.refreshTable("tablename")` to update the metadata, or the caching can be disabled by setting `spark.sql.parquet.cacheMetadata` to `false`.

Example 3-35. Write managed table

```
def saveManagedTable(df: DataFrame): Unit = {
  df.write.saveAsTable("pandas")
}
```

Unless specific conditions are met, the result saved to a Hive managed table will be saved in a Spark specific format that other tools may not be able to understand.

RDDs

Spark SQL DataFrames can easily be converted to RDDs of `Row` objects, and can also be created from RDDs of `Row` objects as well as JavaBeans, Scala case classes, and tuples. For RDDs of strings in JSON format, you can use the methods discussed in "JSON" on page 61. Datasets of type T can also easily be converted to RDDs of Type T, which can provide a useful bridge for DataFrames to RDDs of concrete case classes instead of `Row` objects. RDDs are a special case data source, since when going to/from

RDDs, the data remains inside of Spark without writing out to or reading from an external system.

Converting a DataFrame to an RDD is a transformation (not an action); however, converting an RDD to a DataFrame or Dataset may involve computing (or sampling some of) the input RDD.

Creating a DataFrame from an RDD is not free in the general case. The data must be converted into Spark SQL's internal format.

When you create a DataFrame from an RDD Spark SQL needs to add schema information. If you are creating the DataFrame from an RDD of case classes or plain old Java objects (POJOs), Spark SQL is able to use reflection to automatically determine the schema. You can also manually specify the schema for our data using the structure discussed in "Basics of Schemas" on page 43. This can be especially useful if some of our fields are not nullable. You must specify the schema yourself if Spark SQL is unable to determine the schema through reflection, such as an RDD of `Row` objects (perhaps from calling `.rdd` on a DataFrame to use a functional transformation).

Example 3-36. Creating DataFrames from RDDs

```
def createFromCaseClassRDD(input: RDD[PandaPlace]) = {
  // Create DataFrame explicitly using sqlContext and schema inference
  val df1 = sqlContext.createDataFrame(input)

  // Create DataFrame using sqlContext implicits and schema inference
  val df2 = input.toDF()

  // Create a Row RDD from our RDD of case classes
  val rowRDD = input.map(pm => Row(pm.name,
    pm.pandas.map(pi => Row(pi.id, pi.zip, pi.happy, pi.attributes))))

  val pandasType = ArrayType(StructType(List(
    StructField("id", LongType, true),
    StructField("zip", StringType, true),
    StructField("happy", BooleanType, true),
    StructField("attributes", ArrayType(FloatType), true))))

  // Create DataFrame explicitly with specified schema
  val schema = StructType(List(StructField("name", StringType, true),
    StructField("pandas", pandasType)))
```

```
  val df3 = sqlContext.createDataFrame(rowRDD, schema)
}
```

Case classes or JavaBeans defined inside another class can sometimes cause problems. If your RDD conversion is failing, make sure the case class being used isn't defined inside another class.

Converting a DataFrame to an RDD is incredibly simple; however, you get an RDD of `Row` objects. Since a row can contain anything, you need to specify the type (or cast the result) as you fetch the values for each column in the row. With Datasets you can directly get back an RDD templated on the same type, which can make the conversion back to a useful RDD much simpler.

While Scala has many implicit conversions for different numeric types, these do not generally apply in Spark SQL, instead we use explicit casting.

Example 3-37.

```
def toRDD(input: DataFrame): RDD[RawPanda] = {
  val rdd: RDD[Row] = input.rdd
  rdd.map(row => RawPanda(row.getAs[Long](0), row.getAs[String](1),
    row.getAs[String](2), row.getAs[Boolean](3), row.getAs[Array[Double]](4)))
}
```

If you know that the schema of your DataFrame matches that of another, you can use the existing schema when constructing your new DataFrame. One common place where this occurs is when an input DataFrame has been converted to an RDD for functional filter and then back.

Local Collections

Much like with RDDs, you can also create DataFrames from local collections and bring them back as local collections. The same memory requirements apply; namely, the entire contents of the DataFrame will be in-memory in the driver program. As such, distributing local collections is normally limited to unit tests, or joining small data sets with larger distributed datasets.

Example 3-38. Creating from a local collection

```
def createFromLocal(input: Seq[PandaPlace]) = {
  sqlContext.createDataFrame(input)
}
```

The LocalRelation's API we used here allows us to specify a schema in the same manner as when we are converting an RDD to a DataFrame.

In pre-1.6 versions of PySpark, schema inference only looked at the first record.

Collecting data back as a local collection is more common and often done post aggregations or filtering on the data. For example, with ML pipelines collecting the coefficents or in our Goldlilocks example collecting the quantiles to the driver. For larger data sets, saving to an external storage system (such as a database or HDFS) is recommended.

Just as with RDDs do not collect large DataFrames back to the driver. For Python users, it is important to remember that `toPandas()` collects the data locally.

Example 3-39. Collecting the Result locally

```
def collectDF(df: DataFrame) = {
  val result: Array[Row] = df.collect()
  result
}
```

Additional Formats

As with core Spark, the data formats that ship directly with Spark only begin to scratch the surface of the types of systems with which you can interact. Some vendors publish their own implementations, and many are published on Spark Packages. As of this writing there are over twenty formats listed on the Data Source's page with the

most popular being Avro, Redshift, CSV [1], and a unified wrapper around 6+ databases called deep-spark.

Spark packages can be included in our application in a few different ways. During the exploration phase (e.g. using the shell) you can include them by specifying `--packages` on the command line, as in Example 3-40. The same approach can be used when submitting our application with `spark-submit`, but this only includes the package at run time, not at compile time. For including at compile time you can add the maven coordinates to our builds, or, if building with sbt, the sbt-spark-package plugin simplifies package dependencies with `spDependencies`.

Example 3-40. Starting Spark shell with CSV support

```
./bin/spark-shell --packages com.databricks:spark-csv_2.10:1.3.0
```

Example 3-41. Include spark-csv as an sbt dependency

```
  "com.databricks" % "spark-csv_2.10" % "1.3.0",
```

Once you've included the package with our Spark job you need to specify the format, as you did with the Spark provided ones. The name should be mentioned in the packages documentation. For spark-csv you would specify a format string of `com.databricks.spark.csv`.

There are a few options if the data format you are looking for isn't directly supported in either Spark or one of the libraries. Since many formats are available as Hadoop input formats, you can try to load our data as a Hadoop input format and convert the resulting RDD as discussed in "RDDs" on page 65. This approach is relatively simple, but means Spark SQL is unable to push down operations to our data store [2].

For a deeper integration you can implement our data source using the Data Source API . Depending on which operations you wish to support operator push-down for, your base relation you will need to implement additional traits from the `org.apache.spark.sql.sources` package. The details of implementing a new Spark SQL data source are beyond the scope of this book, but if you are interested the scaladoc for `org.apache.spark.sql.sources` and spark-csv's CsvRelation can be a good ways to get started.

1 There is also a proposal to include csv directly in Spark since it is such a popular format.

2 For example, only reading the required partitions when a filter matches one of our partitioning schemes

Save Modes

In core Spark, saving RDDs always requires that the target directory does not exist, which can make appending to existing tables challenging. With Spark SQL, you can specify the desired behavior when writing out to a path that may already have data. The default behaviour is `SaveMode.ErrorIfExists`; matching RDDs behaviour, Spark will throw an exception if the target already exists. The different save modes and their behaviours are listed in Table 3-11.

Table 3-11. Save modes

Save Mode	Behaviour
ErrorIfExists	Throws an exception if the target already exists. If target doesn't exist write the data out.
Append	If target already exists, append the data to it. If the data doesn't exist write the data out.
Overwrite	If the target already exists, delete the target. Write the data out.
Ignore	If the target already exists, silently skip writing out. Otherwise write out the data.

Example 3-42. Specify save mode of append

```
def writeAppend(input: DataFrame): Unit = {
  input.write.mode(SaveMode.Append).save("output/")
}
```

Partitions (Discovery and Writing)

Partition data is an important part of Spark SQL since it powers one of the key optimizations to allow reading only the required data, discussed more in "Logical and Physical Plans" on page 77. If you know how our downstream consumers may access our data (e.g. reading data based on zip code), when you write your data it is beneficial to use that information to partition our output. When reading the data, it's useful to understand how partition discovery functions, so you can have a better understanding of whether our filter can be pushed down.

Filter push down can make a huge difference when working with large data sets by allowing Spark to only access the subset of data required for your computation instead of doing effectively a full table scan.

When reading partitioned data, you point Spark to the root path of your data, and it will automatically discover the different partitions. Not all data types can be used as partition keys; currently only strings and numeric data are the supported types.

If our data is all in a single DataFrame, the `DataFrameWriter` API makes it easy to specify the partition information while you are writing the data out. The `partitionBy` function takes a list of columns to partition the output on. You can also manually save our separate DataFrames (say if you are writing from different jobs) with individual `save` calls.

Example 3-43. Save partitioned by zip code

```
def writeOutByZip(input: DataFrame): Unit = {
  input.write.partitionBy("zipcode").format("json").save("output/")
}
```

In addition to splitting the data by a partition key, it can be useful to make sure the resulting file sizes are reasonable, especially if the results will be used downstream by another Spark job. See ??? for general guidance.

Datasets

Datasets are an exciting extension of the DataFrame API which provide additional compile time type checking. Datasets can be used when our data can be encoded for Spark SQL and you know the type information at compile time. The Dataset API is a strongly typed collection with a mixture of relational (DataFrame) and functional (RDD) transformations. Like DataFrames, Datasets are represented by a logical plan the Catalyst optimizer can work with and when cached the data is stored in Spark SQL's internal encoding format.

The Dataset API is new in Spark 1.6 and will change in future versions. Users of the Dataset API are advised to treat it as a "preview." Up-to-date documentation on the Dataset API can be found in the scaladoc.

Interoperability with RDDs, DataFrames, and Local Collections

Datasets can be easily converted to/from DataFrames and RDDs, but in the initial version they do not directly extend either. Converting to/from RDDs involves encoding/decoding the data into a different form. Converting to/from DataFrames is almost "free" in that the underlying data does not need to be changed, only extra compile time type information is added/removed.

It is intended that future versions of DataFrames may extend `Dataset[Row]` to allow for easier interoperability.

To convert a DataFrame to a Dataset you can use the `as[ElementType]` function on the DataFrame to get a `Dataset[ElementType]` back as shown in Example 3-44. The `ElementType` must be a case class, or similar such as tuple, consisting of types Spark SQL can represent (see "Basics of Schemas" on page 43). To create `Datasets` from local collections, `createDataSet(...)` on the SQLContext and the `toDS()` implicit are provided on Seqs in the same manner as `createDataFrame(...)` and `toDF()`. For converting from RDD to Dataset you can first convert from RDD to DataFrame and then convert it to a Dataset.

For loading data into a Dataset, unless a special API is provided by your data source, you can first load your data into a DataFrame and then convert it to a Dataset. Since the conversion to the Dataset simply adds information you do not have the problem of eagerly evaluating, and future filters and similar operations can still be pushed down to the data store.

Example 3-44. Create a Dataset from a DataFrame

Converting from a Dataset back to an RDD or DataFrame can be done in similar ways as when converting DataFrames. The `toDF` simply copies the logical plan used in the Dataset into a DataFrame - so you don't need to do any schema inference or conversion as you do when converting from RDDs. Converting a Dataset of type `T` to an RDD of type `T` can be done by calling `.rdd`, which unlike calling `toDF`, does involve converting the data from the internal SQL format to the regular types.

Example 3-45. Convert Dataset to DataFrame & RDD

```
/**
 * Illustrate converting a Dataset to an RDD
 */
def toRDD(ds: Dataset[RawPanda]): RDD[RawPanda] = {
  ds.rdd
}

/**
 * Illustrate converting a Dataset to a DataFrame
 */
def toDF(ds: Dataset[RawPanda]): DataFrame = {
  ds.toDF()
}
```

Compile Time Strong Typing

One of the reasons to use Datasets over traditional DataFrames is their compile time strong typing. DataFrames have runtime schema information but lack compile time

information about the schema. This strong typing is especially useful when making libraries, because you can more clearly specify the requirements of your inputs and your return types.

Easier Functional (RDD "like") Transformations

One of the key advantages of the Dataset API is easier integration with custom Scala and Java code. Datasets expose `filter`, `map`, `mapPartitions`, and `flatMap` with similar function signatures as RDDs, with the notable requirement that our return ElementType also be understandable by Spark SQL (such as tuple or case class of types discussed in "Basics of Schemas" on page 43).

Example 3-46. Create a Dataset from a DataFrame

```
def fromDF(df: DataFrame): Dataset[RawPanda] = {
  df.as[RawPanda]
}
```

Beyond functional transformations, such as `map` and `filter`, you can also intermix relational and grouped/aggregate operations.

Relational Transformations

Datasets introduce a typed version of `select` for relational style transformations. When specifying an expression for this you need to include the type information. You can add this information by calling `as[ReturnType]` on the expression/column.

Example 3-47. Simple relational select on Dataset

```
def squishyPandas(ds: Dataset[RawPanda]): Dataset[(Long, Boolean)] = {
  ds.select($"id".as[Long], ($"attributes"(0) > 0.5).as[Boolean])
}
```

Some operations, such as select, have both typed and untyped implementations. If you supply a `Column` rather than a `TypedColumn` you will get a DataFrame back instead of a Dataset.

Multi-Dataset Relational Transformations

In addition to single Dataset transformations, there are also transformations for working with multiple Datasets. The standard set operations, namely `intersect`, `union`, and `subtract`, are all available with the same standard caveats as discussed in

Table 3-9. Joining Datasets is also supported, but to make the type information easier to work with, the return structure is a bit different than traditional SQL joins.

Grouped Operations on Datasets

Similar to grouped operations on DataFrames, `groupBy` on Datasets return a `GroupedDataset`, on which you can specify your aggregate functions, along with a functional `mapGroups` API. As with the expression in "Relational Transformations" on page 73, you need to use typed expressions so the result can also be a Dataset. Taking our previous example of computing the maximum panda size by zip in Example 3-18, you would rewrite it to be as shown in Example 3-48.

The convenience functions found on `GroupedData` (e.g. `min`, `max`, etc.) are missing, so all of our aggregate expressions need to be specified through `agg`.

Example 3-48. Compute the max panda size per zip code typed

```
def maxPandaSizePerZip(ds: Dataset[RawPanda]): Dataset[(String, Double)] = {
  ds.groupByKey(rp => rp.zip).agg(max("attributes(2)").as[Double])
}
```

Beyond applying typed SQL expressions to aggregated columns, you can also easily use arbitrary Scala code with `mapGroups` on grouped data as shown in Example 3-49. This can save us from having to write custom user defined aggregate functions (UDAFs) (discussed in "Extending with User Defined Functions & Aggregate Functions (UDFs, UDAFs)" on page 74). While custom UDAFs can be painful to write, they may be able to give better performance than `mapGroups` and can also be used on `DataFrames`.

Example 3-49. Compute the max panda size per zip code using map groups

```
def maxPandaSizePerZipScala(ds: Dataset[RawPanda]): Dataset[(String, Double)] = {
  ds.groupByKey(rp => rp.zip).mapGroups{ case (g, iter) =>
    (g, iter.map(_.attributes(2)).reduceLeft(Math.max(_, _)))
  }
}
```

Extending with User Defined Functions & Aggregate Functions (UDFs, UDAFs)

User defined functions and user defined aggregate functions provide you with ways to extend the DataFrame & SQL APIs with your own custom code while keeping the

Catalyst optimizer. The Dataset API is another performant option for much of what you can do with UDFs and UDAFs. This is quite useful for performance, since otherwise you would need to convert the data to an RDD (and potentially back again) to perform arbitrary functions, which is quite expensive. UDFs and UDAFs can also be accessed from inside of regular SQL expressions, making them accessible to analysts or others more comfortable with SQL.

When using UDFs or UDAFs written in non-JVM languages, such as Python, it is important to note that you lose much of the performance benefit, as the data must still be transferred out of the JVM.

If most of your work is in Python but you want to access some UDFs without the performance penalty, you can write your UDFs in Scala and register them for use in Python (as done in Sparkling Pandas).[1]

Writing non-aggregate UDF for Spark SQL is incredibly simple: you simply write a regular function and register it using `sqlContext.udf().register`. If you are registering a Java or Python UDF you also need to specify our return type.

Example 3-50. String length UDF

```
def setupUDFs(sqlCtx: SQLContext) = {
  sqlCtx.udf.register("strLen", (s: String) => s.length())
}
```

Even with JVM languages UDFs are generally slower than the equivalent SQL expression would be if it exists. Some early work is being done in SPARK-14083 to parse JVM byte code and generate SQL expressions.

Aggregate functions (or UDAFs) are somewhat trickier to write. Instead of writing a regular Scala function, you extend the `UserDefinedAggregateFunction` and implement a number of different functions, similar to the functions one might write for `aggregateByKey` on an RDD, except working with different data structures. While they can be complex to write, UDAFs can be quite performant compared with options like `mapGroups` on `Datasets` or even simply written `aggregateByKey` on `RDDs`.

1 As of this writing, the Sparkling Pandas project development is on-hold but early releases still contains some interesting examples of using JVM code from Python.

You can then either use the UDAF directly on columns or add it to the function registry as you did for the non-aggregate UDF.

Example 3-51. UDAF for computing the average

```
def setupUDAFs(sqlCtx: SQLContext) = {
  class Avg extends UserDefinedAggregateFunction {
    // Input type
    def inputSchema: org.apache.spark.sql.types.StructType =
      StructType(StructField("value", DoubleType) :: Nil)

    def bufferSchema: StructType = StructType(
      StructField("count", LongType) ::
      StructField("sum", DoubleType) :: Nil
    )

    // Return type
    def dataType: DataType = DoubleType

    def deterministic: Boolean = true

    def initialize(buffer: MutableAggregationBuffer): Unit = {
      buffer(0) = 0L
      buffer(1) = 0.0
    }

    def update(buffer: MutableAggregationBuffer,input: Row): Unit = {
      buffer(0) = buffer.getAs[Long](0) + 1
      buffer(1) = buffer.getAs[Double](1) + input.getAs[Double](0)
    }

    def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
      buffer1(0) = buffer1.getAs[Long](0) + buffer2.getAs[Long](0)
      buffer1(1) = buffer1.getAs[Double](1) + buffer2.getAs[Double](1)
    }

    def evaluate(buffer: Row): Any = {
      buffer.getDouble(1) / buffer.getLong(0)
    }
  }
  // Optionally register
  val avg = new Avg
  sqlCtx.udf.register("ourAvg", avg)
}
```

This is a little more complicated than our regular UDF, so let's take a look at what the different parts do. You start by specifying what the input type is, then you specify the schema of the buffer you will use for storing the in-progress work. These schemas are specified in the same way as DataFrame and Dataset schemas, discussed in "Basics of Schemas" on page 43.

From there the rest of the functions are implementing the same functions you use when writing `aggregateByKey` on an RDD, but instead of taking arbitrary Scala objects you work with `Row` and `MutableAggregationBuffer`. The final `evaluate` function takes the `Row` representing the aggregation data and returns the final result.

UDFs, UDAFs, and Datasets all provide ways to intermix arbitrary code with Spark SQL.

Query Optimizer

Catalyst is the Spark SQL query optimizer, which is used to take the query plan and transform it into an execution plan that Spark can run. Much as our transformations on RDDs build up a DAG, as you apply relational and functional transformations on DataFrames/Datasets, Spark SQL builds up a tree representing our query plan, called a logical plan. Spark is able to apply a number of optimizations on the logical plan and can also choose between multiple physical plans for the same logical plan using a cost-based model.

Logical and Physical Plans

The logical plan you construct through transformations on DataFrames/Datasets (or SQL queries) starts out as an unresolved logical plan. Much like a compiler, the Spark optimizer is multi-phased and before any optimizations can be performed, it needs to resolve the references and types of the expressions. This resolved plan is referred to as the logical plan, and Spark applies a number of simplifications directly on the logical plan, producing an optimized logical plan. These simplifications can be written using pattern matching on the tree, such as the rule for simplifying additions between two literals. The optimizer is not limited to pattern matching, and rules can also include arbitrary Scala code.

Once the logical plan has been optimized, Spark will produce a physical plan. The physical plan stage has both rule-based and cost-based optimizations to produce the optimal physical plan. One of the most important optimizations at this stage is predicate pushdown to the data source level.

Code Generation

As a final step, Spark may also apply code generation for the components. Code generation is done using Janino to compile Java code. Earlier versions used Scala's Quasi Quotes [1], but the overhead was too high to enable code generation for small datasets.

1 Scala Quasi Quotes are part of Scala's macro system.

In some TPCDS queries, code generation can result in >10x improvement in performance.

In some early versions of Spark for complex queries, code generation can cause failures. If you are on an old version of Spark and run into an unexpected failure it can be worth disabling codegen by setting `spark.sql.codegen` or `spark.sql.tungsten.enabled` to false (depending on version).

Large Query Plans and Iterative algorithms

While the Catalyst optimizer is quite powerful, one of the cases where it currently runs into challenges is with very large query plans. These query plans tend to be the result of iterative algorithms, like graph algorithms or machine learning algorithms. One simple work around for this is converting the data to an RDD and back to DataFrame/Dataset at the end of each iteration - although if your in Python be sure to use the underlying Java RDD rather than round tripping through Python, see ??? for how to do this. Another, somewhat more heavy option, is to write the data to storage and continue from there.

Example 3-52. Round trip through RDD to cut query plan

```
val rdd = df.rdd
rdd.cache()
sqlCtx.createDataFrame(rdd, df.schema)
```

This issue is being tracked in SPARK-13346 and you can see the workaround used in GraphFrames.

JDBC/ODBC Server

Spark SQL provides a JDBC server to allow external tools, such as business intelligence GUIs like tableau, to work with data accessible in Spark and to share resources. Spark SQL's JDBC server requires that Spark be built with Hive support.

Since the server tends to be long lived and runs on a single context, it can also be a good way to share cached tables between multiple users.

Spark SQL's JDBC server is based on the HiveServer2 from Hive, and most corresponding connectors designed for HiveServer2 can be used directly with Spark SQL's JDBC server. Simba also offers specific drivers for Spark SQL.

The server can either be started from the command line or started using an existing HiveContext. The command line start and stop commands are `./sbin/start-thriftserver.sh` and `./sbin/stop-thriftserver.sh`. When starting from the command line, you can configure the different Spark SQL properties by specifying `--hiveconf property=value` on the command line. Many of the rest of the command line parameters match that of `spark-submit`. The default host and port is `localhost:10000` and can be configured with `hive.server2.thrift.port` and `hive.server2.thrift.bind.host`.

When starting the JDBC server using an existing `HiveContext`, you can simply update the config properties on the context instead of specifying command line parameters.

Example 3-53. Start JDBC server on a different port

```
./sbin/start-thriftserver.sh --hiveconf hive.server2.thrift.port=9090
```

Example 3-54. Start JDBC server on a different port in Scala

```
sqlContext.setConf("hive.server2.thrift.port", "9090")
HiveThriftServer2.startWithContext(sqlContext)
```

When starting the JDBC server on an existing `HiveContext` make sure to shutdown the JDBC server when exiting.

Conclusion

The considerations for using DataFrames/Datasets over RDDs are complex and changing with the rapid development of Spark SQL. One of the cases where Spark SQL can be difficult to use is when the number of partitions needed for different parts of our pipeline changes, or if you otherwise wish to control the partitioner. While RDDs lack the Catalyst optimizer and relational style queries, they are able to work with a wider variety of data types and provide more direct control over certain types of operations. DataFrames and Datasets also only work with a restricted subset of data types - but when our data is in one of these supported classes the performance

improvements of using the Catalyst optimizer provide a compelling case for accepting those restrictions.

DataFrames can be used when you have primarily relational transformations, which can be extended with UDFS when necessary. Compared to RDDs, DataFrames benefit from the efficient storage format of Spark SQL, the Catalyst optimizer, and the ability to perform certain operations directly on the serialized data. One drawback to working with DataFrames is that they are not strongly typed at compile time, which can lead to errors with incorrect column access and other simple mistakes.

Datasets can be used when you want a mix of functional and relational transformations while benefiting from the optimizations for DataFrames and are, therefore, a great alternative to RDDs in many cases. As with RDDs, Datasets are parameterized on the type of data contained in them, which allows for strong compile time type checking but requires that you know our data type at compile time (although `Row` or other generic type can be used). The additional type safety of Datasets can be beneficial even for applications that do not need the specific functionality of DataFrames. One potential drawback is that the Dataset API is continuing to evolve, so updating to future versions of Spark may require code changes.

Pure RDDs work well for data that does not fit into the Catalyst optimizer. RDDs have an extensive and stable functional API, and upgrades to newer versions of Spark are unlikely to require substantial code changes. RDDs also make it easy to control partitioning, which can be very useful for many distributed algorithms. Some types of operations, such as multi-column aggregates, complex joins, and windowed operations, can be daunting to express with the RDD API. RDDs can work with any Java or Kryo serializable data, although the serialization is more often more expensive and less space efficient than the equivalent in DataFrames/Datasets.

Now that you have a good understanding of Spark SQL, it's time to continue on to joins, for both RDDs and Spark SQL.

CHAPTER 4

Joins (SQL & Core)

Joining data is an important part of many of our pipelines, and both Spark core and SQL support the same fundamental types of joins. While joins are very common and powerful, they warrant special performance consideration as they may require large network transfers or even create data sets beyond our capability to handle.[1]. In core Spark it can be more important to think about the ordering of operations, since the DAG optimizer, unlike the SQL optimizer isn't able to re-order or push down filters.

Core Spark Joins

In this section we will go over the RDD type joins. Joins in general are expensive since they require that corresponding keys from each RDD are located at the same partition so that they can be combined locally. If the RDDs do not have known partitioners, they will need to be shuffled so that both RDDs share a partitioner and data with the same keys lives in the same partitions, as shown in Figure 4-1. If they have the same partitioner, the data may be colocated, as in Figure 4-3, so as to avoid network transfer. Regardless of if the partitioners are the same, if one (or both) of the RDDs have a known partitioner only a narrow dependency is created, as in Figure 4-2. As with most key-value operations, the cost of the join increases with the number of keys and the distance the records have to travel in order to get to their correct partition.

1 As the saying goes, the cross product of big data and big data is an out of memory exception.

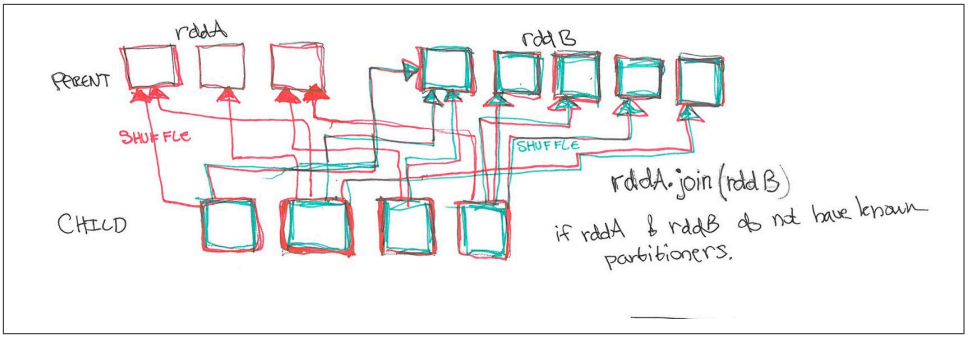


Figure 4-1. Shuffle join

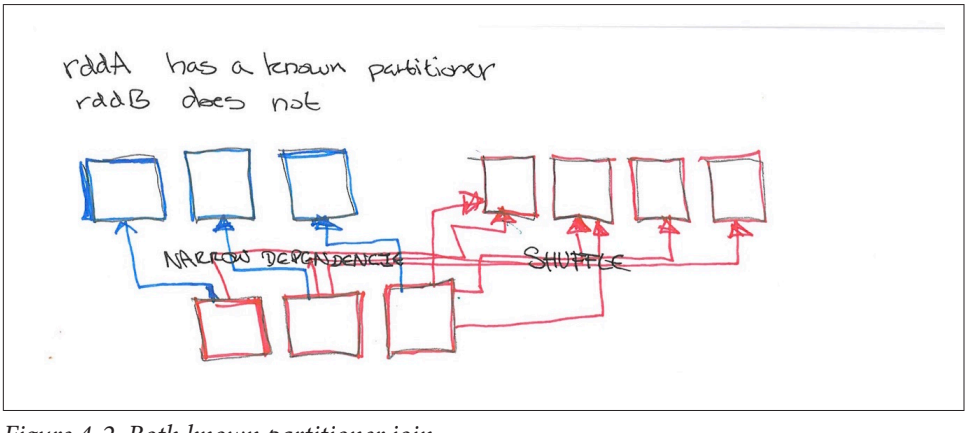


Figure 4-2. Both known partitioner join

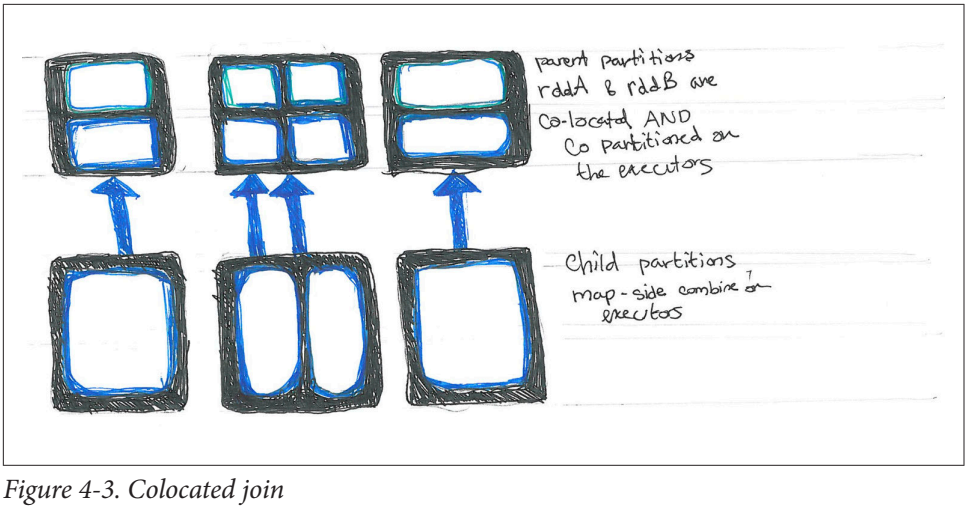


Figure 4-3. Colocated join

Two RDDs will be colocated if they have the same partitioner and were shuffled as part of the same action.

Core Spark joins are implemented using the `coGroup` function. We discuss `coGroup` in "Co-Grouping" on page 147.

Choosing a Join Type

The default join operation in Spark includes only values for keys present in both RDDs, and in the case of multiple values per key, provides all permutations of the key/value pair. The best scenario for a standard join is when both RDDs contain the same set of distinct keys. With duplicate keys, the size of the data may expand dramatically causing performance issues, and if one key is not present in both RDDs you will loose that row of data. Here are a few guidelines:

1. When both RDDs have duplicate keys, the join can cause the size of the data to expand dramatically. It may be better to perform a `distinct` or `combineByKey` operation to reduce the key space or to use `cogroup` to handle duplicate keys instead of producing the full cross product. By using smart partitioning during the combine step, it is possible to prevent a second shuffle in the join (we will discuss this in detail later).
2. If keys are not present in both RDDs you risk losing your data unexpectedly. It can be safer to use an outer join, so that you are guaranteed to keep all the data in either the left or the right RDD, then filter the data after the join.
3. If one RDD has some easy-to-define subset of the keys, in the other you may be better off filtering or reducing before the join to avoid a big shuffle of data, which you will ultimately throw away anyway.

Join is one of the most expensive operations you will commonly use in Spark, so it is worth doing what you can to shrink your data before performing a join.

For example, suppose you have one RDD with some data in the form (Panda id, score) and another RDD with (Panda id, address), and you want to send each Panda some mail with her best score. You could join the RDDs on id and then compute the best score for each address. Like this:

Example 4-1. Basic RDD join

```
def joinScoresWithAddress1( scoreRDD : RDD[(Long, Double)],
 addressRDD : RDD[(Long, String )]) : RDD[(Long, (Double, String))]= {
   val joinedRDD = scoreRDD.join(addressRDD)
    joinedRDD.reduceByKey( (x, y) => if(x._1 > y._1) x else y )
}
```

However, this is probably not as fast as first reducing the score data, so that the first dataset contains only one row for each Panda with her best score, and then joining that data with the address data.

Example 4-2. Pre-filter before join

```
def joinScoresWithAddress2( scoreRDD : RDD[(Long, Double)],
  addressRDD : RDD[(Long, String )]) : RDD[(Long, (Double, String))]= {
  //stuff
 val bestScoreData = scoreRDD.reduceByKey((x, y) => if(x > y) x else y)
 bestScoreData.join(addressRDD)

}
```

If each Panda had 1000 different scores then the size of the shuffle we did in the first approach was 1000 times the size of the shuffle we did with this approach!

If we wanted to we could also perform a left outer join to keep all keys for processing even those missing in the right RDD by using `leftOuterJoin` in place of `join`. Spark also has `fullOuterJoin` and `rightOuterJoin` depending on which records we wish to keep. Any missing values are `None` and present values are `Some('x')`.

Example 4-3. Basic RDD left outer join

```
def outerJoinScoresWithAddress( scoreRDD : RDD[(Long, Double)],
 addressRDD : RDD[(Long, String )]) : RDD[(Long, (Double, Option[String]))]= {
   val joinedRDD = scoreRDD.leftOuterJoin(addressRDD)
    joinedRDD.reduceByKey( (x, y) => if(x._1 > y._1) x else y )
}
```

Choosing an Execution Plan

In order to join data, Spark needs the data that is to be joined (i.e. the data based on each key) to live on the same partition. The default implementation of join in Spark is a *shuffled hash join*. The shuffled hash join ensures that data on each partition will contain the same keys by partitioning the second dataset with the same default partitioner as the first, so that the keys with the same hash value from both datasets are in the same partition. While this approach always works, it can be more expensive than necessary because it requires a shuffle. The shuffle can be avoided if:

1. Both RDDs have a known partitioner.
2. One of the datasets is small enough to fit in memory, in which case we can do a broadcast hash join (we will explain what this is later).

Note that if the RDDs are colocated the network transfer can be avoided, along with the shuffle.

Speeding Up Joins by Assigning a Known Partitioner

If you have to do an operation before the join that requires a shuffle, such as `aggregateByKey` or `reduceByKey`, you can prevent the shuffle by adding a hash partitioner with the same number of partitions as an explicit argument to the first operation and persisting the RDD before the join. You could make the example in the previous section even faster, by using the partitioner for the address data as an argument for the `reduceByKey` step.

Example 4-4. Known partitioner join

```
def joinScoresWithAddress3( scoreRDD : RDD[(Long, Double)],
 addressRDD : RDD[(Long, String )]) : RDD[(Long, (Double, String))]= {
 //if addressRDD has a known partitioner we should use that,
 //otherwise it has a default hash parttioner, which we can reconstrut by getting the number of
 // partitions.
 val addressDataPartitioner = addressRDD.partitioner match {
  case (Some(p)) => p
  case (None) => new HashPartitioner(addressRDD.partitions.length)
 }
 val bestScoreData = scoreRDD.reduceByKey(addressDataPartitioner, (x, y) => if(x > y) x else y)
 bestScoreData.join(addressRDD)
}
```

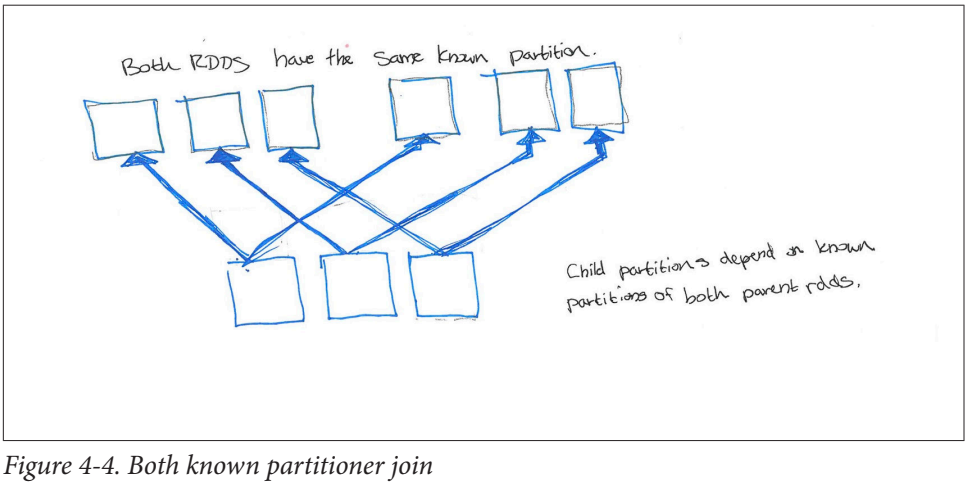


Figure 4-4. Both known partitioner join

Always persist after re-partitioning.

Speeding Up Joins Using a Broadcast Hash Join

A broadcast hash join pushes one of the RDDs (the smaller one) to each of the worker nodes. Then it does a map-side combine with each partition of the larger RDD. If one of your RDDs can fit in memory or can be made to fit in memory it is always beneficial to do a broadcast hash join, since it doesn't require a shuffle. Sometimes (but not always) Spark will be smart enough to configure the broadcast join itself. You can see what kind of join Spark is doing using the `toDebugString()` function.

Example 4-5. toDebugString() on a join

```
scoreRDD.join(addressRDD).toDebugString
```

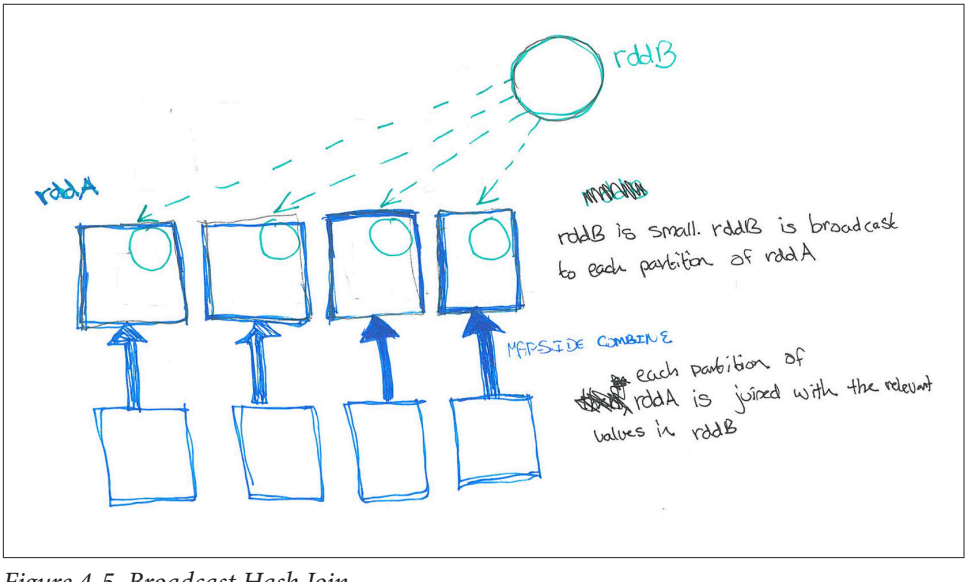


Figure 4-5. Broadcast Hash Join

Partial Manual Broadcast Hash Join

Sometimes not all of our smaller RDD will fit into memory, but some keys are so over-represented in the large data set, so you want to broadcast just the most common keys. This is especially useful if one key is so large that it can't fit on a single

partition. In this case you can use `countByKeyApprox` [1] on the large RDD to get an approximate idea of which keys would most benefit from a broadcast. You then filter the smaller RDD for only these keys, collecting the result locally in a HashMap. Using `sc.broadcast` you can broadcast the HashMap so that each worker only has one copy and manually perform the join against the HashMap. Using the same HashMap you can then filter our large RDD down to not include the large number of duplicate keys and perform our standard join, unioning it with the result of our manual join. This approach is quite convoluted but may allow you to handle highly skewed data you couldn't otherwise process.

Spark SQL Joins

Spark SQL supports the same basic join types as core Spark, but the optimizer is able to do more of the heavy lifting for you - although you also give up some of our control. For example, Spark SQL can sometimes push down or re-order operations to make our joins more efficient. On the other hand, you don't control the partitioner for DataFrames or Datasets, so you can't manually avoid shuffles as you did with core Spark joins.

DataFrame Joins

Joining data between DataFrames is one of the most common multi-DataFrame transformations. The standard SQL join types are all supported and can be specified as the `joinType` in `df.join(otherDf, sqlCondition, joinType)` when performing a join. As with joins between RDDs, joining with non-unique keys will result in the cross product (so if the left table has R1 and R2 with key1 and the right table has R3 and R5 with key1 you will get (R1, R3), (R1, R5), (R2, R3), (R2, R5)) in the output. While we explore Spark SQL joins we will use two example tables of pandas, Example 4-6 and Example 4-7.

While self joins are supported, you must alias the fields you are interested in to different names beforehand, so they can be accessed.

1 If the number of distinct keys is too high, you can also use `reduceByKey`, sort on the value, and take the top k.

Example 4-6. Table of pandas and sizes (our left DataFrame)

Name	Size
Happy	1.0
Sad	0.9
Happy	1.5
Coffee	3.0

Example 4-7. Table of pandas and zip codes (our right DataFrame)

Name	Zip
Happy	94110
Happy	94103
Coffee	10504
Tea	07012

Spark's supported join types are inner, left_outer (aliased as "outer"), left_anti right_outer, full_outer, and left_semi [1]. With the exception of "left_semi" these join types all join the two tables, but they behave differently when handling rows that do not have keys in both tables.

The "inner" join is both the default and likely what you think of when you think of joining tables. It requires that the key be present in both tables, or the result is dropped as shown in Example 4-8 and inner join table.

Example 4-8. Simple inner join

```
// Inner join implicit
df1.join(df2, df1("name") === df2("name"))
// Inner join explicit
df1.join(df2, df1("name") === df2("name"), "inner")
```

1 The "*"s are optional and can be left out. We use them in our examples because we think it is easier to read with the "*"s present

Table 4-1. Inner join of df1, df2 on name

Name	Size	Name	Zip
Coffee	3.0	Coffee	10504
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103
Happy	1.0	Happy	94110
Happy	1.0	Happy	94103

Left outer joins will produce a table with all of the keys from the left table, and any rows without matching keys in the right table will have null values in the fields that would be populated by the right table. Right outer joins are the same, but with the requirements reversed.

Example 4-9. Left outer join

```
// Left outer join explicit
df1.join(df2, df1("name") === df2("name"), "left_outer")
```

Table 4-2. Left outer join df1, df2 on name

Name	Size	Name	Zip
Sad	0.9	null	null
Coffee	3.0	Coffee	10504
Happy	1.0	Happy	94110
Happy	1.0	Happy	94103
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103

Example 4-10. Right outer join

```
// Right outer join explicit
df1.join(df2, df1("name") === df2("name"), "right_outer")
```

Table 4-3. Right outer join df1, df2 on name

Name	Size	Name	Zip
Coffee	3.0	Coffee	10504
Happy	1.0	Happy	94110
Happy	1.0	Happy	94103
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103
null	null	Tea	07012

To keep all records from both tables you can use the full outer join.

Table 4-4. Full outer join df1, df2 on name

Name	Size	Name	Zip
Sad	0.9	null	null
Coffee	3.0	Coffee	10504
Happy	1.0	Happy	94110
Happy	1.0	Happy	94103
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103
null	null	Tea	07012

Left semi joins and left anti joins are the only kind of joins which only has values from the left table. A left semi join is the same as filtering the left table for only rows with keys present in the right table. The left anti join also only returns data from the left table, but instead only returns records which are not present in the right table.

Example 4-11. Left semi join

```
// Left semi join explicit
df1.join(df2, df1("name") === df2("name"), "left_semi")
```

Table 4-5. Left semi join

Name	Size
Coffee	3.0
Happy	1.0
Happy	1.5

Table 4-6. Left anit join

Name	Size
Sad	0.9

Self Joins

Self joins are supported on DataFrames; but we end up with duplicated columns names. So that you can access the results, you need to alias the DataFrames to different names - otherwise you will be unable to select the columns due to name collision. Once you've aliased each DataFrame, in the result you can access the individual columns for each DataFrame with `dfName.colName`.

Example 4-12. Self join

```
val joined = df.as("a").join(df.as("b")).where($"a.name" === $"b.name")
```

Broadcast Hash Joins

In SparkSQL you can see the type of join being performed by calling `queryExecution.executedPlan`. As with core Spark, if one of the tables is much smaller than the other you may want a broadcast hash join. You can hint to Spark SQL that a given DF should be broadcast for join by calling `broadcast` on the DataFrame before joining it (e.g. `df1.join(broadcast(df2), "key")`). Spark also automatically uses the `spark.sql.conf.autoBroadcastJoinThreshold` to determine if a table should be broadcast.

Dataset Joins

Joining Datasets is done with `joinWith`, and this behaves similarly to a regular relational join, except the result is a tuple of the different record types as shown in Example 4-13. This is somewhat more awkward to work with after the join, but also does make self joins, as shown in Example 4-14, much easier, as you don't need to alias the columns first.

Example 4-13. Joining two Datasets

```
val result: Dataset[(RawPanda, CoffeeShop)] = pandas.joinWith(coffeeShops,
  $"zip" === $"zip")
```

Example 4-14. Self join a Dataset

```
val result: Dataset[(RawPanda, RawPanda)] = pandas.joinWith(pandas,
  $"zip" === $"zip")
```

Using a self join and a `lit(true)`, you can produce the cartesian product of your Dataset, which can be useful but also illustrates how joins (especially self-joins) can easily result in unworkable data sizes.

As with DataFrames you can specify the type of join desired (e.g. inner, left_outer, right_outer, leftsemi), changing how records present only in one Dataset are handled. Missing records are represented by null values, so be careful.

Conclusion

Now that you have explored joins, it's time to focus on transformations and the performance considerations associated with them. For those interested in continuing learning more about Spark SQL, we will continue with Spark SQL tuning in ???, where we include more details on join-specific configurations like number of partitions and join thresholds.

CHAPTER 5

Effective Transformations

Most commonly, Spark programs are structured on RDDs: they involve reading in data from stable storage into the RDD format, performing a number of computations and data transformations on the RDD, and writing the result RDD to stable storage or collecting to the driver. Thus, most of the power of Spark comes from its transformations, operations that are defined on RDDs and return RDDs.

At present, Spark contains specialized functionality for about a half-dozen types of RDDs, each with their own properties and scores of different transformation functions. In this section, we hope to give you the tools to think about how your RDD transformation, or series of transformations, will be evaluated. In particular, what kinds of RDDs these transformations return, whether persisting or checkpointing RDDs between transformations will make your computation more efficient, and how a given series of transformations could be executed in the most performant way possible.

The transformations in this section apply to RDDs, which are the abstraction used in Spark Core (and MLlib). RDDs are also used inside of DStreams with Spark Streaming, but they have different functionality and performance properties. Likewise, most of the functions discussed in this chapter are not yet supported in DataFrames and since SparkSQL has a different optimizer, not all of the conceptual lessons of this chapter will carry over to the Spark SQL world.

As Spark moves forward more RDD transformations will become available on Datasets, which can be used in Spark SQL, and which are discussed in "Datasets" on page 71.

Narrow vs. Wide Transformations

In Chapter 2, we introduced one important distinction between types of transformations, those with *wide dependencies* and those with *narrow dependencies*. This distinction is important, because it has strong implications for how transformations are evaluated and, consequently, their performance. In this subsection, we will define the distinct between wide and narrow transformations more precisely, demonstrate how to determine whether a transformation is wide or narrow, and explain why this distinction matters for evaluation, and consequently, performance.

Recall that Spark is lazily evaluated, meaning that transformations are not executed until an action that depends on that transformation is called. This, as we disused in detail in "Lazy Evaluation" on page 23, has important consequences for fault tolerance, performance, and debugging. If the information in this tip is confusing, please refer back to Chapter 2 which will give you the basic understanding of the Spark execution engine needed for this chapter.

To summarize what we covered in Chapter 2, wide transformations are those which require a shuffle, narrow transformations are those that do not. In "Wide vs. Narrow Dependencies" on page 29 we explained that in narrow transformations the child partitions (the partitions in the resulting RDD) depend on a known subset of the parent partitions. While this definition is correct, it is less precise than the formal definition of narrow transformations.

The 2012 paper first presenting the evaluation semantics for Spark defines transformations with narrow dependencies as those in which "each partition of the parent RDD is used by at most one partition of the child RDD" and transformations with wide dependencies as transformations in which "multiple child partitions may depend on [each partition in the parent]." This definition states what we explained in Chapter 2 in reverse; it defined narrow and wide dependencies in terms of the parent RDD not the child. I think the definition presented in Chapter 2 is easier to conceptualize, since we usually design a program thinking from the input data to the output data. However, the Spark evaluation engine (the "DAG"), builds an execution plan in reverse: from the output (the last action) to the input RDD.

The Spark creators' definition is more precise because it mirrors the way Spark is evaluated and rules out the case of one parent partition having multiple children in a narrow dependency. In particular it explains why coalesce is only a narrow transformation when it is reducing rather than increasing the number of partitions. The original definition also makes clear why the number of tasks used to complete a computation corresponds to each output partition rather than each input partition; when RDDs are evaluated the tasks needed to compute a transformation are computed on the child partitions.

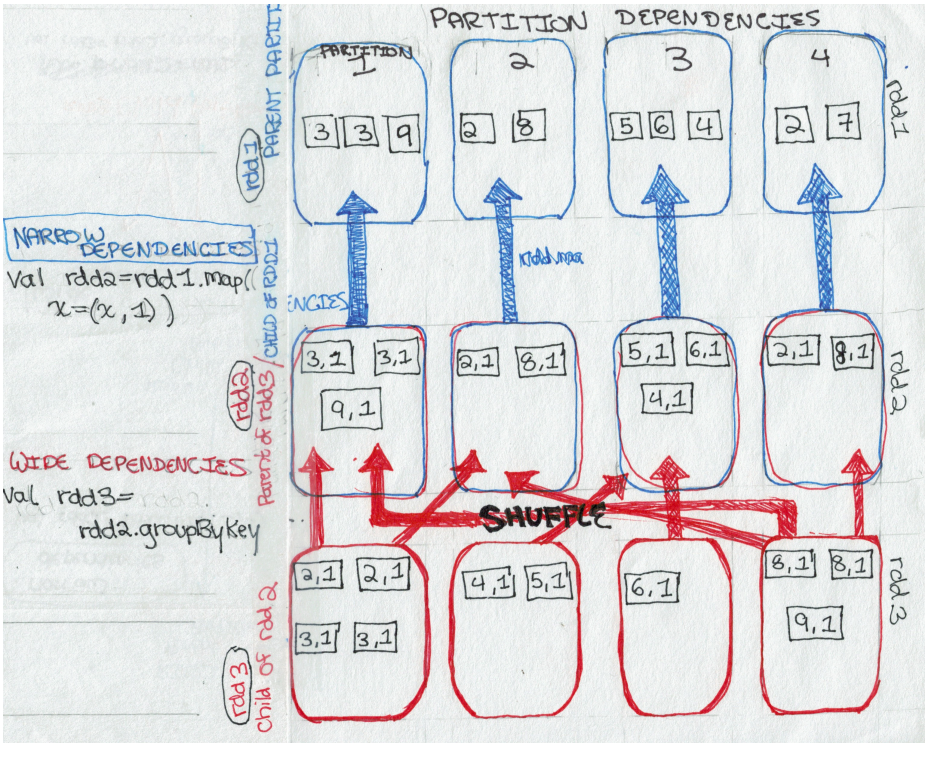
In the following diagram Figure and example Example 5-1, we show dependencies between parent and child partitions for narrow and wide transformations for the following Spark program. Assume `RDD1` is an RDD of integers.

Example 5-1. Narrow vs Wide example

```
//Narrow dependency. Map the rdd to tuples  of (x, 1)
val rdd2 = rdd1.map(x => (x, 1))
//wide dependency groupByKey
val rdd3 = rdd2.groupByKey()
```

We assume the RDD has four partitions. Unlike the diagrams presented in Chapter 2, in these diagrams we will show how actual records in a very small RDD might be distributed amongst the partitions. In this case we show how `RDD1`, `RDD2` and `RDD3` would be partitioned if `RDD1` was an RDD of the integers `3,3,9,2,8,5,6,7`.

We use the same structure as in the diagrams presented in Figure 2-2 and Figure 2-3.Each child partition has arrows pointing to the parent partitions upon which it depends. Blue arrows represent narrow dependencies and red arrows represent wide dependencies.



As you can see, to compute the `map` step, each child partition depends on just one parent, since the data doesn't need to be moved between partitions for the operation to be computed. However in the `groupByKey` step, Spark needs to move an arbitrary number of the records so that those with the same key are in the same partition in order to combine records corresponding to a single key into one iterator (recall that iterator is a local, rather than a distributed collection) Thus, the child partitions depend on many partitions in the parent RDD.

This diagram is intended to show how partitions, an abstract concept used in Spark evaluation, depend on each other, rather than any physical data movement across machines. Each line of squares in the digram represents the same executors at different points in time. The arrows denote dependencies between partitions. In fact repartitioning data does not necessarily require data movement across machines, since partitions may reside on the same executor. When changing the partition of a record does require data movement between executors, the records, have to be passed through the driver rather than transferred directly between the executors.

Implications for Performance

In "Wide vs. Narrow Dependencies" on page 29 we asserted that transformations with narrow dependencies are faster to execute partly because they can be combined and executed in one pass of the data and more easily than transformations with wide dependencies. In this section we hope to explain why this is from an evaluation perspective.

Narrow dependencies don't require data to be moved across partitions, consequently narrow transformations don't require communication with the driver node, and an arbitrary number of narrow transformations can be executed on any subset of the data (any partition) given one set of instructions from the driver. In Spark terminology, we say that each series of narrow transformations can be computed in the same "stage" of the query execution plan.

In contrast, as we stated in "The Anatomy of a Spark Job" on page 34, a shuffle associated with a wide dependency, marks a new stage in the RDDs evaluation. Because tasks must be computed on a single partition and the data needed to compute each partition of a wide dependency may be spread across machines, transformations with wide dependencies may require data to be moved across partitions. Thus, the downstream computations cannot be computed before the shuffle finishes.

For example, it should be intuitive that sorting cannot be accomplished with narrow transformations, because sorting requires an order to be defined on all of the data, not just within each partition. Indeed, the `sortByKey` function has wide dependencies. It requires the data to be partitioned, so all the keys within a certain range live on

the same partition, that way sorting the data on each partition leads to a sorted result. Sorting on each machine, and any other narrow transformations following the sort cannot be done until after the shuffle completes, since the records on each partition may change.

Stage boundaries have important performance consequences. Except in the case of multiple RDD operations like `join`, the stages associated with one RDD must be executed in sequence (see Chapter 4). Thus, not only are shuffles expensive since they require data movement and potential disk I/O (for the shuffle files), they also limit parallelization.

Implications for Fault Tolerance

The cost of failure for a partition with wide dependencies is much higher than for one with narrow dependencies, since it requires more partitions to be re-computed. If one partition in the parent of a `mappedRDD` (the resulting RDD type of a `map` operations) fails, for example, only one of its children must be re-computed, and the tasks needed to recompute that child partition can be distributed across the executors to make this re-computation faster. In contrast, if the parent of the sorted RDD loses a partition, it is possible (in the worst case) that all the child partitions will need to be recomputed. For this reason, the cost of recomputing a partition in the case of failure for a partition with wide dependencies is much higher than for a partition with narrow dependencies.

Chaining together transformations with wide dependencies, particularly if they have a high probability of causing memory errors, only increases the risk of a very expensive re-computation. In some instances, the cost of re-computation may be high enough that it is worth `checkpointing` an RDD, so the intermediate results are saved. We will discuss checkpointing in detail in "Reusing RDDs" on page 117.

The Special Case of Coalesce

The `coalesce` operation is use to change the number of partitions in an RDD. As shown by the diagram in Figure 2-2 in Chapter 2, when `coalesce` reduces the number of output partitions, each parent partition is used in exactly one child partition since the child partitions are the union of several parents. Thus, according to our definition of narrow dependencies `coalesce` is a narrow transformation even though it changes the number of partitions in the RDD. Since tasks are executed on the child partition, the number of tasks executed in a stage that includes a `coalesce` operation is equivalent to the number of partitions in the result RDD of the `coalesce` transformation.

Using `coalesce`, the number of partitions can decrease in one stage without causing a shuffle. However, `coalesce` causes the upstream partitions in the entire stage to execute with the level of parallelism assigned by `coalesce`. Avoid this behavior at the cost of a shuffle by setting the shuffle argument of `coalesce` to true or by using repartition.

However, when `coalesce` increase the number of partitions, each parent partition necessarily depends on several child partitions. Thus, according the our new definition of wide dependencies, using `coalesce` to increase the number of partitions requires a shuffle. The `coalesce` function prioritizes evenly distributing the data across the child partitions. Thus it makes sense that the location of records in the output depends on how many records are stored on each input partition, which cannot be determined during design time.

What Type of RDD does Your Transformation Return?

RDDs are an abstracted concept in two ways: they are can be of almost any arbitrary type of record (e.g. `String`, `Row`, `Tuple`) and they could be members of different implementations of the `RDD` interface with different properties. Different transformations return different implementations of the RDD interface with different properties, and some transformations can only be applied to RDDs with certain record types. Understanding what record type an RDD has can be important for performance and surprisingly difficult as Spark programs get complicated. In this section we will discuss preserving record type information. Understanding the data locality and partitioning information is associated with the resulting RDD of a transformation can help avoid unnecessary shuffles. We will discuss this in detail in "Preserving Partitioning Information Across Transformations" on page 151

The RDD is a collection type which, much like collection types in Scala, Java and most other strongly typed languages, are instantiated with a type parameter indicating the type of the members in the collection. In Scala, the syntax for a type parameter is brackets e.g. `Seq[String]` indicates a sequence of String objects. RDDs are similarly typed. For example, if you used `sc.textfile` to read in your RDD, you will end up with an RDD of `String` type.

The RDDs record type information is important because many transformations are only defined on RDDs that are of a particular type, so trying to use methods on an RDD of generic type will return compile time or runtime errors. For example, if an RDD of tuples has lost its type information and is interpreted by the compiler to be of type `RDD[Any]` or even `RDD[(Any, Int)]`, calling `sortByKey` will not compile. Since `sortByKey` can only be called on RDDs of key/value pairs where the keys have some

implicit ordering. Similarly, numeric functions such as `max`, `min`, and `sum` can only be called on RDDs of `Long`, `Int`, or `Double`.

Record type information is one of many places in the Spark API where implicit conversions are likely to cause difficulties. If you are writing sub-routines to be used in RDD transformation it can be easier to specify the helper function's input and return types concretely and often simpler not to write them on a generic type.

One instance in which I have run into problems with loosing type information is when working with DataFrames as RDDs. DataFrames can be implicitly converted to RDDs of `Rows`. However, since the SparkSQL `Row` object is not strongly typed (they can be created from sequences of any value type), the Scala compiler cannot "remember" the type of value used to create the row. Indexing a row will return a value of type `Any`, which must be cast to a more specific type, such as a `String` or an `Int`, to perform most calculations. The type information for the rows is in the schema. However, converting to an RDD throws away a DataFrame's schema information, so it is important to bind the DataFrame schema to a variable. One of the advantages of the Dataset API is that it is a strongly typed, so the values in each row will retain their type information even after conversion to an RDD.

Minimizing Object Creation

"Garbage collection" is the process of freeing up the memory allocated for an object once that object is no longer needed. Since Spark runs in the JVM, which has automatic memory management and large data structures, garbage collection can quickly become an expensive part of our job and garbage collection or "GC" errors are a common cause of failure. Even if garbage collection overhead doesn't prohibit a job from running, garbage collection creates additional serialization time which can significantly slow it down. We can minimize the GC cost by reducing the number of objects and the size of those objects. We can reduce the size and number of our objects by reusing existing objects, and using data structures, such as primitive types, that take up less space in memory.

Reusing Existing Objects

Some RDD transformations, allow us to modify the parameters in the lambda expression rather than returning a new object. For example, in the sequence function of the aggregation function for `aggregateByKey` and `aggregate`, we can modify the original accumulator argument, and define the combine function in such a way so that the combination is created by modifying the first of the two accumulators. A common and effective paradigm for complicated aggregations is to define a Scala class with sequence and combine operations that return the existing object using the `this.type` annotations.

For example, suppose that we wanted to do some custom aggregation that is not already defined in Spark. Lets say, that we have an RDD of key/value pairs where the keys are the Panda instructors and the values are the Pandas' report cards. For each instructor we want the length of the longest word used, the average number of words used per report card, and the number of instances of the word happy. One pretty good, easy to read approach would be to use the `aggregateByKey` function, which takes three arguments a zero value, that represents an empty accumulator, a sequence function which takes the accumulator and a value and adds the value to the accumulator and a combine operator which defines how the accumulators should be combined. In this instance we could define our accumulator to be an object with four fields: the total count of all the words, the total number of reports, the longest word seen so far, and the total number of mentions of the word happy.

For clarity we can define this as its own object with methods for sequence and combine. I have named this object: `MetricsCalculator` and it might be coded as follows.

Example 5-2. Aggregation example without object reuse

```
class MetricsCalculator(
 val totalWords : Int,
 val longestWord: Int,
 val happyMentions : Int,
 val numberReportCards: Int) extends Serializable {

 def sequenceOp(reportCardContent : String) : MetricsCalculator = {
   val words = reportCardContent.split(" ")
   val tW = words.length
   val lW = words.map( w => w.length).max
   val hM = words.count(w => w.toLowerCase.equals("happy"))

   new MetricsCalculator(
     tW + totalWords,
     Math.max(longestWord, lW),
     hM + happyMentions,
     numberReportCards + 1)
 }

  def compOp(other : MetricsCalculator) : MetricsCalculator = {
    new MetricsCalculator(
      this.totalWords + other.totalWords,
      Math.max(this.longestWord, other.longestWord),
      this.happyMentions + other.happyMentions,
      this.numberReportCards + other.numberReportCards)
  }

  def toReportCardMetrics =
    ReportCardMetrics(longestWord, happyMentions, totalWords.toDouble/numberReportCards)
}
```

We would could then use this object in the arguments to our aggregation function in a routine that maps the RDD of instructors and report text to a case class with the three metrics we care about.

Example 5-3. Case class for aggregations

```
case class ReportCardMetrics(longestWord : Int, happyMentions : Int, averageWords : Double)
```

Example 5-4. Second version of Aggregations

```
 class MetricsCalculator(
  val totalWords : Int,
  val longestWord: Int,
  val happyMentions : Int,
  val numberReportCards: Int) extends Serializable {

  def sequenceOp(reportCardContent : String) : MetricsCalculator = {
    val words = reportCardContent.split(" ")
    val tW = words.length
    val lW = words.map( w => w.length).max
    val hM = words.count(w => w.toLowerCase.equals("happy"))

    new MetricsCalculator(
      tW + totalWords,
      Math.max(longestWord, lW),
      hM + happyMentions,
      numberReportCards + 1)
  }

   def compOp(other : MetricsCalculator) : MetricsCalculator = {
     new MetricsCalculator(
       this.totalWords + other.totalWords,
       Math.max(this.longestWord, other.longestWord),
       this.happyMentions + other.happyMentions,
       this.numberReportCards + other.numberReportCards)
   }

   def toReportCardMetrics =
     ReportCardMetrics(longestWord, happyMentions, totalWords.toDouble/numberReportCards)
}
```

This method is far superior to using using three simple map and reduce methods (which would require three shuffles). The aggregate function should consolidate the sequence operators and some of the reducing map side. However, it has the disadvantage of creating a new instance of our custom overhead for each record in the dataset, and for each combine step. A very simple way to reduce the cost of object creation would be to modify our `MetricsCalculator` to use the very common `this.type` design paradigm. So that the sequence operation modifies the original accumulator

and the combine operation modifies the first accumulator rather than returning a new one.

Example 5-5. Aggregation example with object re-use

```
class MetricsCalculator_ReuseObjects(
  var totalWords : Int,
  var longestWord: Int,
  var happyMentions : Int,
  var numberReportCards: Int) extends Serializable {

  def sequenceOp(reportCardContent : String) : this.type = {
    val words = reportCardContent.split(" ")
    totalWords += words.length
    longestWord = Math.max(longestWord, words.map( w => w.length).max)
    happyMentions += words.count(w => w.toLowerCase.equals("happy"))
    numberReportCards +=1
    this
  }

  def compOp(other : MetricsCalculator_ReuseObjects) : this.type = {
    totalWords += other.totalWords
    longestWord = Math.max(this.longestWord, other.longestWord)
    happyMentions += other.happyMentions
    numberReportCards += other.numberReportCards
    this
  }

  def toReportCardMetrics =
    ReportCardMetrics(longestWord, happyMentions, totalWords.toDouble/numberReportCards)
}
```

Our aggregation routine will remain the same.

It should be obvious that the Scala code within the sequence operator is slower than it needs to be. Rather than performing three different functional calls on the `words` array we ought to go through the string as a string buffer, counting the words, keeping track of the longest word, and counting the occurrence of the word happy or at the very least using a while loop to parse the `words` array rather than three recursive calls. I have left this solution since I think it is easier to read and the primary intention of the example is to show how to optimize the `aggregateByKey` Spark routine.

Reduce, which call aggregate and the fold operations, (`foldLeft`, `fold`, `foldRight`) also have this property. However, these aggregation functions are a special case. In general it is best to avoid mutable data structures in Spark code (and Scala code in general) because they can lead to serialization errors and may have inaccurate results.

For many any other RDD functions, particularly narrow transformations, modifying the first value of the argument is not safe because of the way that the expressions are chained together in Lazy evaluation and made be evaluated multiple times. For example if you have an RDD of mutable objects modifying the arrays with a map function may lead to in accurate results since the objects may be re-used more times than you expect especially if the RDD is recomputed.

Using Smaller Data Structures

Spark is a memory hog, and an important way to optimize Spark jobs for both time and space is to stick to primitive types rather than custom classes. Although it may make code less readable, using arrays rather than case classes or tuples can reduce GC overhead. Scala arrays, which are exactly Java arrays under the hood are the most memory efficient of the Scala collection types. Scala tuples are objects, so in some instances it might be better to use a two or three element array rather than a tuple for expensive operations. The Scala collection types in general incur a higher GC overhead than Arrays.

Notice that our ReportCardMetrics object is just a wrapper for a few numeric values. Although it is less clean and less object oriented, it is more space efficient to use a four element array of integers. We can maintain the same readable code paradigm by using a Scala `object` instead of a `class` and defining the sequence and combine operations as functions on strings and arrays as follows: .Using an array as the aggregation object

Example 5-6.

```
object MetricsCalculator_Arrays extends Serializable {
  val totalWordIndex = 0
  val longestWordIndex = 1
  val happyMentionsIndex = 2
  val numberReportCardsIndex = 3

  def sequenceOp(reportCardMetrics : Array[Int],
    reportCardContent : String) : Array[Int] = {

    val words = reportCardContent.split(" ")
    //modify each of the elements in the array
    reportCardMetrics(totalWordIndex) += words.length
    reportCardMetrics(longestWordIndex) = Math.max(reportCardMetrics(longestWordIndex),
      words.map( w => w.length).max)
    reportCardMetrics(happyMentionsIndex) += words.count(w => w.toLowerCase.equals("happy"))
    reportCardMetrics(numberReportCardsIndex) +=1
    reportCardMetrics
  }

  def compOp(x : Array[Int], y : Array[Int]) : Array[Int] = {
```

```
    //combine the first and second arrays by modifying the elements in the first array
    x(totalWordIndex)  += y(totalWordIndex)
    x(longestWordIndex) = Math.max(x(longestWordIndex), y(longestWordIndex))
    x(happyMentionsIndex) += y(happyMentionsIndex)
    x(numberReportCardsIndex) += y(numberReportCardsIndex)
    x
  }

  def toReportCardMetrics(ar : Array[Int]) : ReportCardMetrics =
    ReportCardMetrics(
      ar(longestWordIndex),
      ar(happyMentionsIndex),
      ar(totalWordIndex)/ar(numberReportCardsIndex)
    )
}
```

We would then need to modify our aggregation code slightly: .Aggregation with arrays to minimize expensive object creation

Example 5-7.

```
  def calculateReportCardStatistics_withArrays(rdd : RDD[(String, String)]
  ): RDD[(String, ReportCardMetrics)] = {

    rdd.aggregateByKey(
      //the zero value is a four element array of zeros
      Array.fill[Int](4)(0)
    )(
    //seqOp adds the relevant values to the array
      seqOp = (reportCardMetrics, reportCardText) =>
        MetricsCalculator_Arrays.sequenceOp(reportCardMetrics, reportCardText),
    //combo defines how the arrays should be combinewd
      combOp = (x, y) => MetricsCalculator_Arrays.compOp(x, y))
    .mapValues(MetricsCalculator_Arrays.toReportCardMetrics)
  }
```

Within a function, it is often beneficial to avoid intermediate object creation. It is important to remember that converting between types (such as between different flavors of Scala collections) creates intermediate objects. For example, suppose that observing my note in the previous section you wanted to speed up the sequence function of the `MetricsCalculator_ReuseObjects` object and you realized that your coworker has written a general purpose utility that finds the instances of the word happy and the longest word in a collection of strings. .Function with implicit sequence conversions

Example 5-8.

```
  def findWordMetrics[T <:Seq[String]](collection : T ): (Int, Int)={
    val iterator = collection.toIterator
```

```
    var mentionsOfHappy = 0
    var longestWordSoFar = 0
    while(iterator.hasNext){
      val n = iterator.next()
      if(n.toLowerCase == "happy"){
        mentionsOfHappy +=1
      }
      val length = n.length
      if(length> longestWordSoFar)
        longestWordSoFar = length

    }
    (longestWordSoFar, mentionsOfHappy)
  }
```

Your co-worker helpfully defined her function on any type that extends Scala `Traversable` index. So you won't need to convert the array of words at all and can happily write the following code:

Example 5-9. Aggregation with bad implicit conversions

```
  val totalWordIndex = 0
  val longestWordIndex = 1
  val happyMentionsIndex = 2
  val numberReportCardsIndex = 3
  def fasterSeqOp(reportCardMetrics : Array[Int], content  : String): Array[Int] = {
    val words: Seq[String] = content.split(" ")
    val (longestWord, happyMentions) = CollectionRoutines.findWordMetrics(words)
    reportCardMetrics(totalWordIndex) += words.length
    reportCardMetrics(longestWordIndex) = longestWord
    reportCardMetrics(happyMentionsIndex) += happyMentions
    reportCardMetrics(numberReportCardsIndex) +=1
    reportCardMetrics
  }
```

Unfortunately, in terms of object creation, this code is actually worse than the previous code. It is creating creating an object around the data in your words array twice! Once, when you call the `findWordMetrics` routine, since the input array has to be implicitly converted to a Traversable object and again, when your coworker's code casts the traversable to an iterator.

Modifying a value passed in to your transformation is not always safe, so double check the documentation for the function you are using.

Beyond reducing the objects that are directly allocated, Scala's implicit conversions can sometimes cause additional allocations.

Iterator-to-Iterator Transformations with `mapPartitions`

The RDD `mapPartitions` function takes as its argument a function from an `iterator` of records, representing the records on one partition, to another `iterator` of records representing the output partition.

The `mapPartitions` transformation is one of the most powerful in Spark, since it lets the user define an arbitrary routine on one partition of data. The `mapPartitions` transformation can be used for very simple data transformations like string parsing, but it can also be used for complex, expensive data processing work to solve problems such as secondary sort or highly custom aggregations. Many of Spark's other transformations, like `filter`, `map`, and `flatMap`, can be built using `mapPartitions`. Optimizing the `mapPartitions` routines is an important part of harnessing the power of Spark as we will see in Chapter 6. To allow Spark flexibility with spilling it's important to represent your functions inside of `mapPartitions` in such a way that your functions don't force loading the entire partition in-memory (e.g. implicitily converting to a list). Iterators have many methods we can write functional-style transformations on, or you can construct your own custom iterator. When a transformation directly takes and returns an iterator without forcing it through another collection, we call these *iterator-to-itarator* transformations.

What Is an Iterator-To-Iterator Transformation?

A Scala `iterator` object is not actually a collection, but a function that defines a process of accessing the elements in a collection one-by-one. Not only are iterators immutable, but the same element in an iterator can only be accessed one time. In other words, iterators can only be traversed once (and they extend the Scala interface `TraversableOnce`). Iterators have some of the same methods defined on them as other immutable Scala collections, such as mappings (`map` and `flatMap`), additions (`++`), folds (`foldLeft`, `reduceRight`, `reduce`), element conditions (`forall` and `exists`) and traversals such as `next` and `foreach`. In some instances, these methods behave differently than other Scala collections. Since the iterator can only be traversed once, any of the iterator methods that require looking at all the elements in the iterator will leave the original iterator empty. Java has its own implementation of iterators, `java.util.Iterator`, which have the same benefits as Scala iterators for Spark's evaluation.

Beware of your function calls. It is easy to accidentally consume an iterator by calling `size` or trigger an implicit conversion.

In some ways it can be helpful to conceptualize iterator methods as we would RDD methods, as either *transformations* or *actions*, because like an RDD, an iterator is a actually a set of evaluation instructions rather than a stored state. Some iterator methods, like `next`, `size`, and `foreach`, traverse the iterator and evaluate it (more like an action). Others, like `map` and `flatMap`, return a new iterator - which is really a set of evaluation instructions - much like RDD transformations return a new RDD. However in contrast to Spark transformations, iterator transformations are executed linearly, one element at at time, rather than in parallel. This makes iterators slower but much easier to use than if they could be executed in parallel. For example, if we needed to, store running some information about the records we have seen we can do that in a filter or a map function since each element mapping happen sequentially. (See the iterator example at the end of this section). One-to-one functions are also not chained together in iterator operations so three `map` calls still requires three passes through the data.

By "iterator-to-iterator transformation" we mean using one of these iterator "transformations" to return a new iterator rather than a) converting the iterator to a different collection or b) by evaluating the iterator with one of the iterator "actions" and building a new collection. To re-iterate, using a while loop to traverse the elements of an iterator and build a new collection (even a new iterator) *does not* qualify as an iterator-to-iterator transformation. Converting an iterator to a more intuitive collection type, manipulating it and converting back to an iterator is a not an iterator-to-iterator transformation. Indeed, converting an the iterator argument in `mapPartitions` to a collection object eliminates all the benefits of iterator-to-iterator transformations.

Iterators can be converted to any other Scala collection type. However, converting them requires accessing each of the elements. Thus, after it has been converted to a new collection type, an iterator will be at its last element (empty). Beware of implicit conversions, they are a good way to loose all the data in your iterator and force a whole partition to be loaded into memory.

Space and Time Advantages

The primary advantage of using iterator-to-iterator transformations in Spark routines is that they transformations allow Spark to selectively spill data to disk. Conceptually, an iterator-to-iterartor transformation means defining a process for evaluating ele-

ments one at a time. Thus, Spark can apply that procedure to batches of records rather than reading an entire partition into memory or creating a collection with all of the output records in-memory and then returning it. Consequently iterator-to-iterator transformations allow Spark to manipulate partitions that are two large to fit in memory on a single executor without out of memory errors.

Furthermore, keeping the partition as an iterator allow Spark to use disk space more selectively. Rather than spilling an entire partition when it doesn't fit in memory, the iterator-to-iterator transformation allows Spark to spill only those records which do not fit in memory saving disk i/o and the cost of re-computation. Using methods defined on iterators avoids defining intermediary data structures, either by building new collections to return or through conversions between collection types. Reducing the number of large intermediate data structures is a way to avoid unnecessary object creation which can slow down garbage collection as we talked about in "Minimizing Object Creation" on page 99.

Unfortunately the Spark Streaming `mapPartitions` API is one of relatively few places where the Scala API decisively out performs the Java one. Prior to Spark 1.6, `mapPartitions` in Spark Streaming was defined on objects type Java `Iterable` rather than Java `Iterator` and thus automatically reads the entire collection into memory. In the Spark Core, the Java API still uses `Iterable` rather than iterators as the grouped result of `groupByKey` eliminating the possibility of using an iterator-to-itarator transformation to map grouped data.

An Example

For all their advantages, iterators can be a much harder abstraction to conceptualize and use than collection types such as arrays and Hash Maps, which users my be more familiar with from other languages. Here we provide an example of a complicated `mapPartitions` routine which given a sorted RDD of `(value, columnIndex), count)` tuples and a list of rank statistics on this partition returns the `(value, columnIndex)` pairs that represent ranks statistics. This method is part of the optimal solution to the "Goldilocks problem" which is presented in full in "Back to Goldilocks (Again)" on page 168 and introduced in "The Goldilocks Example" on page 130.

Example 5-10. Example `mapPartitions`

```
private def findTargetRanksIteratively(
        sortedAggregatedValueColumnPairs : RDD[((Double, Int), Long)],
        ranksLocations : Array[(Int, List[(Int, Long)])]): RDD[(Int, Double)] = {

  sortedAggregatedValueColumnPairs.mapPartitionsWithIndex((partitionIndex : Int,
```

```
    aggregatedValueColumnPairs : Iterator[((Double, Int), Long)]) => {

    val targetsInThisPart: List[(Int, Long)] = ranksLocations(partitionIndex)._2
   if (targetsInThisPart.nonEmpty) {
     FindTargetsSubRoutine.asIteratorToIteratorTransformation(aggregatedValueColumnPairs,
       targetsInThisPart)
    }
    else Iterator.empty
  })
}
```

This routine is a good example of a place where we are likely to see performance gains from an iterator-to-iterator transformation, since it is a complicated routine performed on partitions that we anticipate will be too large to fit in memory. However it is an instance where using iterators is, from a design perspective, a non-obvious choice, because we have to keep a map of running totals with the number of elements for each column we have seen so far. A more straightforward way to design this routine would be as follows: Loop through the iterator, store the running totals in a `hashMap`, and build a new collection of the elements we want to keep using an array buffer, then convert the array buffer to an iterator.

Example 5-11. Map Partitions example without an iterator-to-iterator transformation

```
def withArrayBuffer(valueColumnPairsIter : Iterator[((Double, Int), Long)],
  targetsInThisPart: List[(Int, Long)] ): Iterator[(Int, Double)] = {

    val columnsRelativeIndex: Predef.Map[Int, List[Long]] =
      targetsInThisPart.groupBy(_._1).mapValues(_.map(_._2))

  //the column indices of the pairs that are desired rank statistics that live in this partition.
    val columnsInThisPart: List[Int] = targetsInThisPart.map(_._1).distinct

  //a HashMap with the running totals of each column index. As we loop through the iterator
  //we will update the hashmap as we see elements of each column index.
    val runningTotals : mutable.HashMap[Int, Long]=  new mutable.HashMap()
    runningTotals ++= columnsInThisPart.map(columnIndex => (columnIndex, 0L)).toMap

  //we use an array buffer to build the resulting iterator
    val result: ArrayBuffer[(Int, Double)] =
    new scala.collection.mutable.ArrayBuffer()

    valueColumnPairsIter.foreach {
      case ((value, colIndex), count) =>

        if (columnsInThisPart contains colIndex) {

          val total = runningTotals(colIndex)
          //the ranks that are contains by this element of the input iterator.
          //get by filtering the
          val ranksPresent =  columnsRelativeIndex(colIndex)
```

```
                        .filter(index => (index <= count + total) && (index > total))

          ranksPresent.foreach(r => result += ((colIndex, value)))

          //update the running totals.
          runningTotals.update(colIndex, total + count)
        }
      }
    //convert
    result.toIterator
  }
```

At first this looks like an okay solution since we are estimating that the number of elements we are returning is small, and because array buffers are usually a relatively performant way to build up Scala collections. However, if the input data is very large relative to the cluster size, we still see out-of-memory errors and failures in this step. As we expect, a more efficient solution would be to use an iterator-to-iterator transformation. We can convert this subroutine to an iterator-to-iterator transformation although our routine is not parallizable (it requires keeping a list of running totals), because it can be completed on one element of the iterator without any information about the other elements. The final solution uses the `filter` function of iterators to eliminate any elements that are not in the final data and a `flatMap`, to build the new iterator of elements in the resulting partitions.

Example 5-12. MapPartitions with iterator-to-iterator transformations

```
def asIteratorToIteratorTransformation(valueColumnPairsIter : Iterator[((Double, Int), Long)],
  targetsInThisPart: List[(Int, Long)] ): Iterator[(Int, Double)] = {

  val columnsRelativeIndex = targetsInThisPart.groupBy(_._1).mapValues(_.map(_._2))
  val columnsInThisPart = targetsInThisPart.map(_._1).distinct

  val runningTotals : mutable.HashMap[Int, Long]=  new mutable.HashMap()
   runningTotals ++= columnsInThisPart.map(columnIndex => (columnIndex, 0L)).toMap

  //filter out the pairs that don't have a column index that is in this part
  val pairsWithRanksInThisPart =  valueColumnPairsIter.filter{
    case (((value, colIndex), count)) =>
      columnsInThisPart contains colIndex
   }

  //map the valueColumn pairs to a list of (colIndex, value) pairs that correspond to one of the
  //desired rank statistics on this partition.
  pairsWithRanksInThisPart.flatMap{

    case (((value, colIndex), count)) =>

        val total = runningTotals(colIndex)
        val ranksPresent: List[Long] = columnsRelativeIndex(colIndex)
```

```
                                        .filter(index => (index <= count + total)
                                          && (index > total))

        val nextElems: Iterator[(Int, Double)] =
          ranksPresent.map(r => (colIndex, value)).toIterator

        //update the running totals
        runningTotals.update(colIndex, total + count)
        nextElems
    }
  }
```

This approach allows the function to spill to disk selectively by working with each element in the iterator one at a time. This implementation saves space by incrementally building the result rather than storing the new collection type in-memory as an array buffer. It saves a penny on garbage collection by not creating the array buffer as an intermediate step.

If you are using an `ArrayBuffer`, to build a new collection for `mapPartitions`, it is always possible, and more performant to use a `map` or `flatMap` on the iterator to incrementally add new elements.

Set Operations

Spark has a variety of set-like operations, some of which are expensive, and some of which have different behavior than the mathematical definitions of the equivalent operations. In this section we hope to explain how to use these operations safely and effectively

Since RDDs aren't distinct, the biggest difference between these and mathematical set operations is how they handle duplicates. For example, `union` merely combines its arguments, so the result of union will always have the size of both RDDs combined. However, `intersection` and `subtract` behave more like familiar set operations, but since the input RDDs can have duplicates the results may be unexpected. Subtracting, will remove all of the elements in the first RDD that have a key present in the second RDD. Thus it is possible that by subtracting, the result will be smaller than the size of the first RDD minus the size of the second, break one of the laws of set theory.

For example, the following simple unit test will pass:

Example 5-13. Subtract example

```
val a = Array(1, 2, 3, 4, 4, 4, 4)
val b = Array(3, 4)
val rddA = sc.parallelize(a)
```

```
val rddB = sc.parallelize(b)
val rddC =  rddA.subtract(rddB)
assert(rddC.count() < rddA.count() - rddB.count())
```

Intersection, co-groups the argument RDDs using their values as keys and filters out those elements that don't appear in both. Consequently the result of intersection contains no duplicates. Although this is the expected behavior for intersection, using several set operations on RDDs containing duplicates can lead to unexpected behavior. The union of the two RDDs in the previous example is an RDD containing two elements, 1 and 2. Thus, we cannot "recreate" `RDDA` as the union of the intersection and the subtraction as the following unit test demonstrates:

Example 5-14. Intersection example

```
val a = Array(1, 2, 3, 4, 4, 4, 4)
val b = Array(3, 4)
val rddA = sc.parallelize(a)
val rddB = sc.parallelize(b)
val intersection =  rddA.intersection(rddB)
val subtraction = rddA.subtract(rddB)
val union = intersection.union(subtraction)
assert(!rddA.collect().sorted.sameElements(union.collect().sorted))
```

To make an RDD more like a set, you can use `distinct` prior to computing any set operations. However calling `distinct` will cause a shuffle if the partitioner is not known.

Reducing Setup Overhead

Some operations require setup work per-worker or per-partition, like creating a database connection or setting up a random number generator. For transformations you can use `mapPartitions`, do the setup work, per partition in the map function, and then perform your desired transformation on the iterator for the partition. We will illustrate doing this with a prng Example 5-15.

Example 5-15. Create one PRNG per partition

```
rdd.mapPartitions{itr =>
  // Only create once RNG per partitions
  val r = new Random()
  itr.filter(x => r.nextInt(10) == 0)
}
```

It is important to remember to use an "Iterator-to-Iterator Transformations with mapPartitions" on page 106 transformation so as to allow spilling to disk selectively.

Beyond using this pattern to reduce setup overhead in transformations, another common pattern is creating a connection inside of an action to save our data. If our work is writing out the data we can use the same pattern as with `mapPartitions` except with `foreachPartition`.

If the setup work can be serialized, a broadcast variable can distribute the object that we cover next. If the setup work can't be serialized, a broadcast variable with a transient lazy val can be used as well. See Example 5-17 in the next section.

Shared Variables

Spark has two types of shared variables, broadcast variables and accumulators, each of which can only be written in one context (driver or worker) and read in the other. Broadcast variables can be written in the driver program and read on the executors, whereas accumulators are written to on the executors and read on the driver.

Broadcast Variables

Broadcast variables give us a way to take a local value on the driver and distribute a read only copy to each machine rather than shipping a new copy with each task. Broadcast variables might not seem especially useful, since we can just capture a local variable in our closure to transfer data from the driver to the workers - but the savings of only sending one copy per machine versus sending one copy per task can make a huge difference, especially when the same broadcast variable is used in additional transformations. Two common examples of using broadcast variables is broadcasting a small table to join against, or broadcasting a machine learning model to be able to run the predictions on our data.

Creating a broadcast variable is done by calling `broadcast` on the Spark Context. This distributes the value to the workers and gives us back a wrapper which allows us to access the value on the workers by calling `value`. If a `broadcast` variable is created with a variable input, the input should not be modified after the variable has been created since existing workers will not see the updates and new workers may see the new value.

Example 5-16. Sample broadcast of a hashset of invalid panda locations to filter out

```
val invalid = HashSet() ++ invalidPandas
val invalidBroadcast = sc.broadcast(invalid)
input.filter{panda => !invalidBroadcast.value.contains(panda.id)}
```

Example 5-17. Create one PRNG per worker

```
class LazyPrng {
  @transient lazy val r = new Random()
}
def customSampleBroadcast[T: ClassTag](sc: SparkContext, rdd: RDD[T]): RDD[T]= {
  val bcastprng = sc.broadcast(new LazyPrng())
  rdd.filter(x => bcastprng.value.r.nextInt(10) == 0)
}
```

The value for a broadcast variable must be a local, serializable value (so no RDDs etc.).

Internally Spark uses broadcast variables for the Hadoop job configuration objects and large blocks of Python code for UDFs. If a broadcast variable is no longer needed, you can explicitly remove it by calling `unpersist()` on the broadcast variable.

Accumulators

Accumulators are the second side of Spark's shared variables, allowing us to collect by-product information from a transformation or action on the workers and bring the result back to the driver. With Spark's execution model, Spark adds to accumulators only once the computation has been triggered (e.g. by an action) and if the computation happens multiple times, Spark will update the accumulator each time. This multiple counting can be desirable for process level information, like computing the entire time spent parsing records. However it can be disastrous for data related information like counting the number of invalid records.

Spark accumulators have had an API update for 2.0 - these examples will be updated in future versions although the underlying implementation remains very similar (except for custom types).

Accumulators can be unpredictable and in their current state are best used where potential multiple counting is the desired behavior.

Accumulators have a number of built-in types that make it easy to create an accumulator for. Accumulators are not intended for collecting large amounts of information, so if you find yourself adding a large number of elements to a collection or appending to a string you may wish to consider a separate action instead of an accumulator. The default operation for numeric accumulators is the + operation, so we could use this to sum the fuzzyness of all of the pandas as shown in Example 5-18.

Example 5-18. Compute fuzzyness of pandas with accumulators

```
def computeTotalFuzzyNess(sc: SparkContext, rdd: RDD[RawPanda]): (RDD[(String, Long)], Double) = {
  val acc = sc.accumulator(0.0) // Create an accumulator with the initial value of 0.0
  val transformed = rdd.map{x => acc += x.attributes(0); (x.zip, x.id)}
  // accumulator still has zero value
  transformed.count() // force evaluation
  // Note: This example is dangerous since the transformation may be evaluated multiple times
  (transformed, acc.value)
}
```

Beyond the off-the-shell types and operations - accumulators support a wide variety of data types provided the operation is associative, but some are more easy to get in trouble with than others. To use an accumulator of a different type, you need to implement the `AccumulatorParam` interface and provide a `zero` and `addInPlace` methods. The `zero` method returns an identity for the provided type, so for integer addition a 0 whereas integer multiplication would be 1. If we wanted to instead compute the result max all of the integers together rather than adding them we could write the following code Example 5-19.

Example 5-19. Compute maximum panda id

```
def computeMaxFuzzyNess(sc: SparkContext, rdd: RDD[RawPanda]): (RDD[(String, Long)], Double) = {
  object MaxDoubleParam extends AccumulatorParam[Double] {
    override def zero(initValue: Double) = initValue
    override def addInPlace(r1: Double, r2: Double): Double = {
      Math.max(r1, r2)
    }
  }
  // Create an accumulator with the initial value of Double.MinValue
  val acc = sc.accumulator(Double.MinValue)(MaxDoubleParam)
  val transformed = rdd.map{x => acc += x.attributes(0); (x.zip, x.id)}
  // accumulator still has Double.MinValue
  transformed.count() // force evaluation
  // Note: This example is dangerous since the transformation may be evaluated multiple times
```

```
    (transformed, acc.value)
  }
```

This still requires that the result is the same as the type we are accumulating. If we wanted to collect all of the distinct elements, we would likely want to collect a set and the types would be different. For this there is the `AccumulableParam` interface we can implement as follows:

Example 5-20. Compute unique panda ids

```
def uniquePandas(sc: SparkContext, rdd: RDD[RawPanda]): HashSet[Long] = {
  object UniqParam extends AccumulableParam[HashSet[Long], Long] {
    override def zero(initValue: HashSet[Long]) = initValue
    // For adding new values
    override def addAccumulator(r: HashSet[Long], t: Long): HashSet[Long] = {
      r += t
      r
    }
    // For merging accumulators
    override def addInPlace(r1: HashSet[Long], r2: HashSet[Long]): HashSet[Long] = {
      r1 ++ r2
    }
  }
  // Create an accumulator with the initial value of Double.MinValue
  val acc = sc.accumulable(new HashSet[Long]())(UniqParam)
  val transformed = rdd.map{x => acc += x.id; (x.zip, x.id)}
  // accumulator still has Double.MinValue
  transformed.count() // force evaluation
  acc.value
}
```

You may provide a name for accumulators in Scala so they show up in the WebUI, just add a name as the second param. This does involve calling `toString` on the accumulator though - so if that is an expensive operation leave your accumulator unnamed.

When working with cached data our accumulators can seem almost consistent, but as discussed in "Interaction with Accumulators" on page 127 this is not the case.

There is a proposal to add data property (or "consistent") accumulators in Spark 2.1 [1] which avoids double counting - but this remains un-merged.

1 Originally planned for 2.0

Internally, starting in Spark 2.0, Spark uses accumulators to keep track of task metrics.

Reusing RDDs

Spark offers several options for RDD re-use including persisting, caching, and checkpointing, although Spark does not perform any of these automatically [1]. Spark does not do so by default since storing RDD for re-use breaks some pipelining and can be a waste if the RDD is only used one or the data is inexpensive to recompute. Persisting (of which caching is one type) requires a lot of space in-memory or on disk and is unlikely to improve performance for operations that are preformed only once. On large datasets memory or disk space necessary for persisting in memory/disk or of storing with off heap persistence or *checkpointing* the RDD (writing the RDD to an external file system) is often higher than the cost of re-computing some partitions in the case of failure or even a whole RDD used multiple times. However for some specific kinds of Spark programs reusing an RDD can lead to huge performance gains, both in the terms of speed and reducing failures.

Cases For Re-Use

In this section we cover some instances when persisting or checkpointing RDDs so that they can be reused, rather than recomputed can lead to performance gains. Broadly speaking the most important cases for re-use are using an RDD many times, performing multiple actions on the same RDD and for long chains of, or very expensive, transformations.

Iterative Computations

For transformations that use the same parent RDD multiple times, reusing and RDD and forcing evaluation of the RDD can help avoid repeated computations. For example, if you were performing a loop of joins to the same dataset, persisting that dataset could lead to huge performance improvements since it ensures that the partitions of that RDD will be available in-memory to do each join.

In the following example we are computing the root mean squared error (RMSE) on a number of different RDDs representing predictions from different models. To do this we have to join each RDD of predictions to an RDD of the data in the validation set.

1 Some notable exceptions are inside of certain ML algorithms, which if passed in an unpersisted RDD will automatically persist and unpersist the RDD

In this example we use `persist()`, which persists the RDD in memory. As we will explain in "Types of Reuse: Cache, Persist, Checkpoint, Shuffle Files" on page 120, `cache()` is equivalent to `persist()`, which is equivalent to `persist("MEMORY_ONLY")`.

Example 5-21. A function with Iterative Computations

```
val testSet: Array[RDD[(Double, Int)]] = Array(validationSet.mapValues(_ + 1), validationSet.mapVa
validationSet.persist() //persist since we are using this RDD several times
val errors = testSet.map( rdd => {
    RMSE(rdd.join(validationSet).values)
})
```

Without persisting, Spark would have to reload and repartition the training dataset RDD to complete the `join`. However with persistence, the training RDD will stayed loaded in-memory on the executors with each run of the algorithm. We discuss performance considerations with different kinds of join in detail in "Core Spark Joins" on page 81.

`Checkpointing`, another form of RDD re-use that writes and RDD to external storage also will break the RDD's lineage although it won't keep the partitions loaded on the executors.

Multiple Actions on the Same RDD

Without re-using an RDD each action called on an RDD will launch its own Spark Job with the full lineage of RDD transformations. Persisting and checkpointing breaks the RDD's lineage, so that the same series of transformations preceding the persist or checkpoint call will only be executed once. Since persisting checkpointing an RDD lasts for the duration of a Spark application, an RDD persisted during one Spark job will be available in a subsequent job executed with the same context. For example, suppose if we wanted to collect the first 10% of the records in an RDD. We could use the following code which calls `sortByKey`, then `count`, then `take`: .An example of two actions without a persist step

Example 5-22.

```
val sorted = rddA.sortByKey()
val count = sorted.count()
val sample: Long = count / 10
sorted.take(sample.toInt)
```

The sort, and presumably the read operation, needed to create the RDD, `sorted`, will occur twice if we do not store the RDD: once in the job called by `count` and again in the job called by `take`. We can't test this element of the execution programmatically, but if you were to run this application and view the web UI you would see that this

code launches two jobs and each one includes a sort stage. However if we add a persist or checkpoint call before the actions, the transformation will only be executed once, since Spark builds a lineage graph from either an RDDs creation or a persisted/checkpointed RDD. .Two Actions with a persist step

Example 5-23.

```
val sorted = rddA.sortByKey()
val count = sorted.count()
val sample: Long = count / 10
rddA.persist()
sorted.take(sample.toInt)
```

Caching and persisting only survive for the duration of an application. To re-use data between Spark Applications use checkpointing.

If the Cost To Compute Each Partition Is Very High

Sometimes, even if a program does not use the same RDD multiple times persisting and checkpointing can speed up a routine and reduce the cost of failures by storing intermediary results. Persisting or checkpointing can be particularly useful if the cost of computing one partition is very high because it insures that in the case of down stream failures, the entire expensive operation will not need to be recomputed. For example, if your program requires a long series of one to one transformations, those transformations will all be combine into very computationally intensive tasks. While this is good so long as the tasks succeed and fit in memory, it does mean that if one of the down stream transformations fails the cost to recompute a single partition may be enormous. If all of the narrow transformations together create more GC overhead or memory strain then your cluster's executors can handle, checkpointing or persisting "off_heap" can be particularly useful since these options allow the RDD to be stored outside of the Spark Executor memory, leaving space to compute. These options are also the only way to prevent recompute if the entire Spark worker fails. Sometimes breaking up a long linage graph for its own sake can help a job succeed since it means each of the tasks will be smaller.

Although narrow transformations are generally faster than wide ones, some individual narrow transformations, such as training a model per partition or working with very wide rows can be expensive. In these cases, reusing an RDD after the expensive computation so it is not recomputed can be important. Similarly, after expensive wide transformations persistence can be useful since it prevents and entire shuffle being recomputed in the event of a single node failure.

Deciding if Re-Compute is Inexpensive Enough

Although persisting is a flag ship feature of Spark, it is not free. Re-using RDDs, when required, will be space intensive to store data in-memory and take time to serialize and de serialize. If persisting causes the data to spill to disk the operation will incur expensive disk I/O. Caching with Java based memory structures (any of Spark's options besides using OFF-HEAP storage options) will incur a much higher garbage collecting cost than recomputing. Checkpointing or persisting to disk has the disadvantages of MapReduce, causing expensive write and read operations. Furthermore, breaking an RDD's lineage prevents transformations with narrow dependencies from being combined into a single tasks, meaning more network traffic as the data has to be read multiple times. For instance persisting or checkpointing between a simple `map` and `filter` step will force evaluation. Causing Spark to do two passes through the data rather than just one, since the transformation has to be evaluated in order to materialize the RDD after the `map`.

My experience has been that it is easy to underestimate just how expensive storing an RDD is relative to recomputing. I have also found that for relatively simple operations the cost of the read operation needed to lord the RDD far outweighs the others, so persisting is most useful when it prevents triggering another read.

The guidelines provided above are good heuristics for when re-use will provide significant benefits. In general, it is worth re-using an RDD rather than recomputing it if the computation is large relative to your cluster and the rest of your job. In many ways, the best way to tell if you need to re-use your RDDs is by experimentation. If your job runs very slowly, one of the first things to try can be persistence. Since Persisting and checkpointing will help reduce the cost of recomputing data in the case of a failure or eliminate it all together. If a job is failing with GC or out of memory errors, checkpointing or persisting OFF HEAP may enable the job to succeed at all. We will go over debugging and cluster sizing information in detail in ??? and ???, respectively.

Types of Reuse: Cache, Persist, Checkpoint, Shuffle Files

If you decide that you need to reuse your RDD, Spark provides a multitude of options for how to store the RDD, so it is important to understand the different problems different types of persistence can solve and how they apply to your use case. There are three primary operations that you can use to store your RDD: cache, persist and checkpoint. In general, persisting and caching are most useful to avoid recomputation during one spark job or to break RDDs with long lineages, since they keep and RDD around on the executors during a Spark Job. Checkpointing is most useful to prevent failures and a high cost of re-computation by saving intermediate

results and like persisting, it helps avoid computation and minimize the cost of failure and avoid re-computation by breaking the linage graph. the cost of failure .

Persist and Cache

Persisting an RDD means materializing an RDD (usually by storing it in-memory on the executors), for re-use *during the current job*. Spark remembers a persisted or persisted RDD's lineage so that it can recompute it for the duration of a Spark job if one of the persisted partitions is lost. After the job ends, The `persist` function takes a `StorageLevel` argument that specifies how the RDD should be stored. Spark provides a number of different storage levels as constants, but each one is created based on five pieces of information. Calling `toString` on a storage level will reveal what options it contains. The Spark documentation about persistenceincludes a fairly comprehensive list of the out of the box storage options which are exposed to you.

Still, we think it may useful to provide some more information about each of the five properties that compose each storage option, since this should give you a deeper understanding of which storage option to choose.

1. `useDisk`: Whether partitions that do not fit in memory should be written to disk, storage level flags containing `DISK` such as `MEMORY_AND_DISK` enable this. By default, if partitions do not fit in memory they will simply be evicted and will need to be recomputed when the persisted RDD is used (see "LRU Caching" on page 125). Thus, persisting to disk can insure that re-computation of those additional large partitions is avoided. However, reading from disk can be time intensive so persistence to disk is only important if the cost of re-computation is particularly high.

It may be beneficial to allow writing to disk if you expect that an RDD cannot fit in memory. However, if the cost of recomputing the partitions is not high (they are simple mappings and don't reduce the size of the data) it may actually be faster to recompute some partitions rather than read from disk.

ONE: `useMemory` Whether the RDD should be stored in-memory or be directly written to disk. The `DISK_ONLY` storage levels are the only options that mark this as false. Most of the speed benefits of caching come from keeping RDDs in memory, so if the motivation for re-use is fast access for repeated computations, it is probably a good idea to choose a storage option that stores partitions in memory. However if memory is a serious issue, or a cluster is noisy and partitions are evicted, this option may be compelling.

TWO: `useOfHeap` Whether the RDD should be stored outside of the Spark executor, in an external system such as, Tachyon. The storage option `OFF_HEAP` uses this option. We will talk more about the benefits of Tachyon in "Tachyon" on page 124.

THREE: `deserialized` Whether the RDD should be stored as deserialized Java objects. As we will discuss in, ???, this can make storing RDDs more space efficient, especially when using a faster serializer, but incurs some performance overhead. Storage options which include `_SER` such as `MEMORY_ONLY_SER` enable serialization.

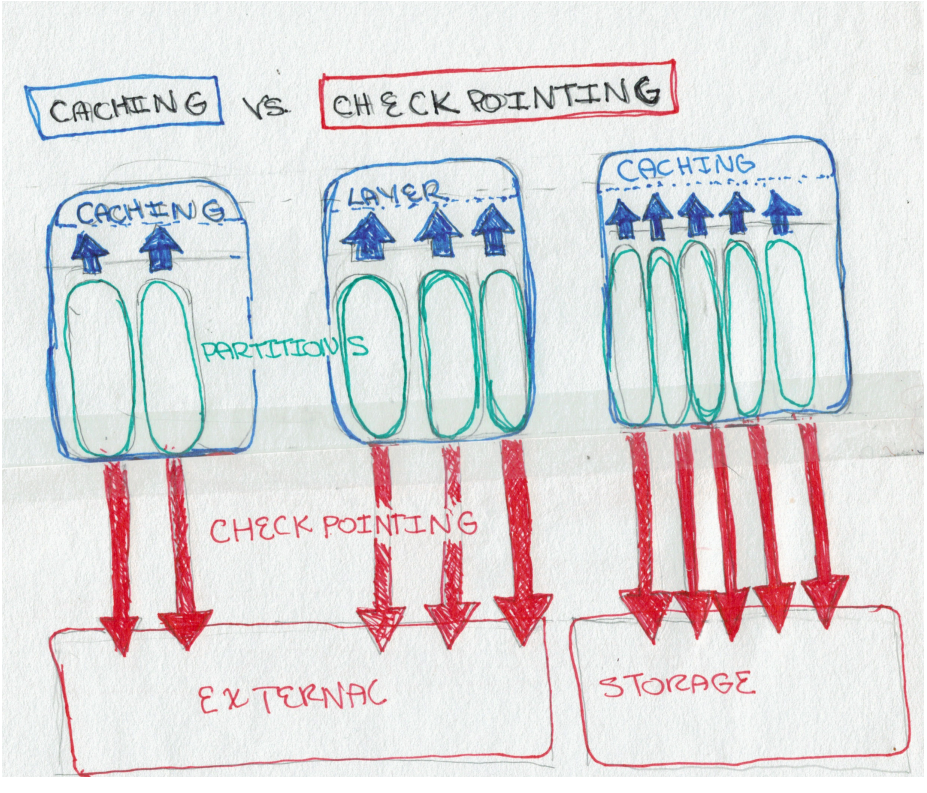
The first thing to try if your RDD is too large to persist in-memory is to try to serialize it with the `MEMORY_ONLY_SER` option since that will keep the RDD fast to access, but decrease the memory needed to store it.

FOUR: `replication`: An integer representing the number of nodes on which each partition should be replicated. By default this is set to 1, however serialization options which end in `_2`such as `DISK_ONLY_2` replicate each partition across two nodes. Use this option to insure faster fault tolerance. However, be aware that it replication incurs double the space and speed costs of persistence without replication. Replication is usually only necessary in an instance of a noisy cluster or bad connection where failures are unusually likely or if you do not have time to recompute in case of failure, such as when serving a live web application.

The RDD operation `cache()` is equivalent to the persist operation with no storage level argument i,e, `persist()`. Both `cache()` and `persist()` persist the RDD with the default storage level `MEMORY_ONLY`, which is equivalent to `StorageLevel(false, true, false, true)`, which stores RDDs in-memory as deserialized Java objects, does not write to disk as partitions get evicted, and doesn't replicate partitions.

Checkpointing

Checkpointing writes the RDD to an external storage system, such as HDFS or S3, and, in contrast to persisting, forgets the RDD's lineage. Since checkpointing requires writing the RDD outside of Spark, checkpointed information survives beyond the duration of a single Spark Application and forces evaluation of an RDD. Checkpointing takes up more space in external storage and may be slower than persisting since it requires potentially costly write operations. However, it does not use any Spark memory and will not incur re-computation if a Spark worker fails.



This diagram illustrates the difference between in-memory persistence and checkpointing and RDD. Persisting stores the RDD's partitions in-memory or on disk in the caching layer of each executor. Checkpointing writes each partition to some external system.

It is best to use checkpointing when the cost of failure and re-computaiton is of more concern than additional space in external storage. Broadly speaking, we advise persisting when jobs are slow, and checkpointing when they are failing. If a Spark job is failing due to out of memory errors, checkpointing will reduce the cost and likelihood of failure without using up memory on the executors. If your jobs are failing due to network errors or pre-emption on a noisy cluster, checkpointing can reduce the likelihood of failure by breaking up a long running job into smaller segments. To call checkpoint, call `setCheckpointDir(directory: String)` from the Spark Context object and pass in a path to a location on HDFS to write the intermediate results. Then, in in the Spark job call `.checkpoint()` from the RDD.

Checkpointing Example

In the following example is used in w a version of one of the functions we will describe in detail in "Back to Goldilocks (Again)" on page 168 that makes use of custom storage level and checkpointing options. In this case we are doing several very expensive transformations. First a sort and then two very substantial map partitions routines. When running on a noisy cluster, we found it advantagious to checkpoint this function after the sort. The value of the `director` variable is the checkpoint directory and `sorted` is a sorted RDD or key / value pairs.

Example 5-24. Checkpoint example

```
def findQuantilesWithCustomStorage(valPairs: RDD[((Double, Int), Long)],
  colIndexList: List[Int],
  targetRanks: List[Long],
  storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK,
  checkPoint : Boolean, directory : String = ""): Map[Int, Iterable[Double]] = {

  val n = colIndexList.last+1
  val sorted  = valPairs.sortByKey()
  if (storageLevel != StorageLevel.NONE)
    sorted.persist(storageLevel)

  if (checkPoint) {
    sorted.sparkContext.setCheckpointDir(directory)
    sorted.checkpoint()
  }

  val partitionColumnsFreq = getColumnsFreqPerPartition(sorted, n)
  val ranksLocations  = getRanksLocationsWithinEachPart(targetRanks, partitionColumnsFreq, n)
  val targetRanksValues = findTargetRanksIteratively(sorted, ranksLocations)
  targetRanksValues.groupByKey().collectAsMap()
}
```

Spark includes a *Local Checkpointing* option that truncates the RDD's lineage graph but doesn't persist to stable storage. This is not suitable for cluster which may experience failures, preemption, or dynamic scale downs during the time the RDD may be referenced.

Tachyon

Tachyon is an distributed, in-memory storage system that is developed separately from Spark. It sits in above a storage system, such as S3 or HDFS and a can be used on its own or with an external computational framework such as Spark or MapReduce. Like Spark, Tachyon can be used in a standalone cluster mode, or with Mesos or Yarn. Read more about Tachyon's architecture and how to integrate it with Spark in the Tachyon documentation

Tachyon can be used as an input or output source for spark applications (data stored with Tachyon can be used to create RDDs) or for `OFF_HEAP` persistence during a Spark application. Using Tachyon for persistence has several advantages. First, it reduces garbage collection overhead, since data is not stored as Java object. Second, it allows multiple executors to share the same externally memory pool in Tachyon. Third, since the data is stored in-memory outside of Spark, it is not lost if individual executors crash. It can be particularly useful if you are seeing GC errors, want to re use an RDD but are running out of memory, or need to reuse very large RDDs between multiple applications.

Tachyon's developer and user communities are very strong in China, so part of its documentation may be stronger in Mandarin than in English.

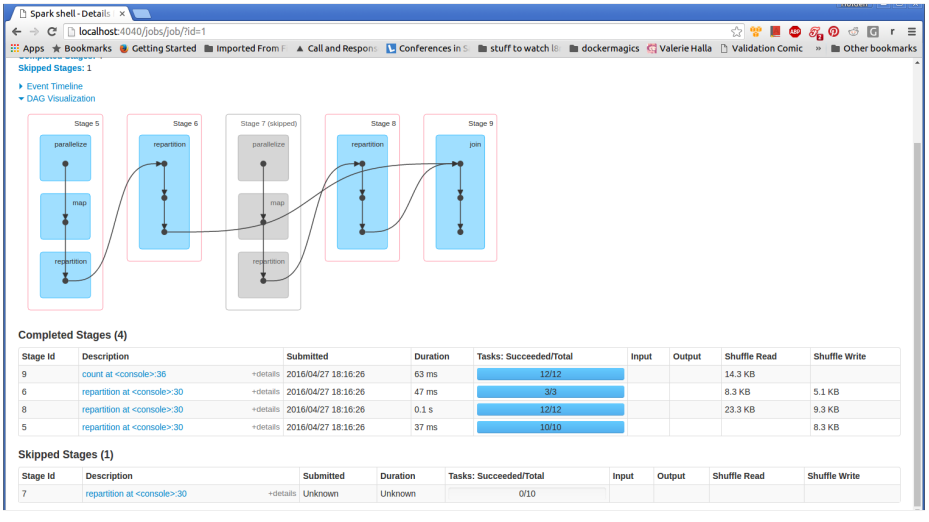
LRU Caching

RDDs which are stored in-memory and/or on disk in Spark are never automatically un-persisted. They stay in-memory for the duration of a Spark application, until the driver program calls the function `unpersist`, or memory/ storage pressure causes their eviction. Spark uses *Least Recently Used* or LRU caching, to determine which partitions to evict if the executors begin to run out of memory.

LRU caching dictates that the data structure that was least recently accessed will be evicted. Because of Lazy evaluation it may be a bit tricky for you as the user to guess which partitions will be evicted first, but generally they are those called by s Spark job that was executed first, or in an earlier stage within a given Spark Job. LRU caching behaves differently for different persistence options. For memory-only persistence operations configured with LRU caching, Spark will recompute the last partition each time it is used. For memory and disk options, LRU caching will write the evicted partition to disk. If you want to take a persisted RDD out of memory to free up space use `unpersist`.

Shuffle Files

One of the less traditional options for re-use is implicit, when shuffling data Spark writes the output to shuffle files on the workers. Unlike the other caches, we can't determine if a given RDD still has it shuffle files present, e.g. there is no equivalent of the `isCheckPointed` command which returns true if that RDD has been checkpointed. In general though, shuffle files aren't explicitly cleaned up until an RDD goes out of scope.



The performance of reusing shuffle files is similar to the performance of an RDD which is cached at the level of disk only.

Shuffle files can be large, and there is no explicit cache management. Keeping references to RDD's depending on shuffled out put can lead to out of disk errors as there is no explicit cache management. This is discussed more in ???.

Noisy Cluster Considerations

Noisy clusters, or those with a high volume of un-predictable traffic, are a fundamental challenge to Spark's evaluation. By default Spark doesn't save most intermediate results (besides in a shuffle step), thus in the case of pre-emptions, Spark will have to recompute the calculation in the job up to the point of failure. In a noisy cluster, where long running jobs are often interrupted this poses a huge challenge. Checkpointing can be especially helpful to get jobs to run at all, since it breaks an RDD's linage, reducing the cost to recompute down stream transformations, and persists to external storage, so that un expected failures do not lead to data loss. If failures are common but not fatal, it may be worthing configuring your job to persist to multiple machines using a storage option like `MEMORY_AND_DISK_2` which replicates data on two machines. That way, failures on one node will not require a recompute. This can be especially important with wide transformations which are very expensive.

By default Spark uses a first-in-first-out (FIFO) system to queue jobs within a system. This means that the first job submitted will run in its entirety, getting priority on all

the available resources while the stages are launching tasks. However if a job doesn't need the whole cluster, the next job may start. This system can be useful to ensure that space-intensive jobs are able to use the resources that they need. However, if your job is queued behind a massive, many hour process the FIFO system can be frustrating. Spark offers a fair scheduler, modeled after the Hadoop fair scheduler, which will allocate the tasks from different jobs to the executors in a "round-robin fashion" i.e. parsing out a few tasks to the executors from each job. With the fair scheduler, a short, small job can be launched while an earlier long running job is still running.

The fair scheduler also supports putting jobs into pools, and allocating different priority (weight) to the pools. Jobs within a pool are allocated the same number or resources, and the pools are allocated resources according to their weight. Using pools can be a good way to ensure that high priority jobs or very expensive jobs are completed, or to insure that users are allocated resources evenly regardless or how many jobs they submit. You can read more about using and configuring a fair scheduler in the the Spark Job Scheduling Documentation

Interaction with Accumulators

The interaction of caching and accumulators can make reasoning about accumulators more difficult. Since as mentioned, our if part of the data is recomputed our accumulator will have the values added to it multiple times. Further more, not all computations will always compute the entirety of a partition. Seemingly caching would fix this, however if a machine fails or if additional data is cached such that part of our data is evicted - this can lead to a recompute. This means that a job that appeared to compute the correct value on small data may later compute the incorrect value on large data.

Conclusion

Now that you have explored how to get the most out of your standard RDD transformations, as well as joins, it's time to explore what additional concerns Key/Value data bring to the mix. Not all of the techniques you will have learned need to be applied in every Spark program, and some of the takeaways from this chapter are more about when certain tools are not a good fit (see accumulators). Many of the same techniques and considerations for standard RDD transformations apply when working with Key/Value data - if your transformation doesn't depend on the key, the techniques from this chapter may even be more relevant.

CHAPTER 6

Working with Key/Value Data

Like any good distributed computing tool, Spark relies heavily on the key/value pair paradigm particularly for wide transformations that require the data to be redistributed between machines. Anytime we want to perform grouped operations in parallel or change the ordering of records amongst machines, be it computing an aggregation statistic or merging customer records, the key/value functionality of Spark is useful as it allows us to easily parallelize our work. Spark has its own `PairRDDFunctions` class with a set of operations defined on RDDs of tuples. The `PairRDDFunctions` class, made available through implicit conversion, contains most of Spark's methods for joining, and custom aggregations. The `OrderedRDDFunctions` class contains the methods for sorting and available to RDDs of tuples in which the first element (the key) has an implicit ordering.

Similar operations are available on Datasets as discussed in "Grouped Operations on Datasets" on page 74

Despite their utility, key/value operations can lead to a number of performance issues that we will discuss in this chapter. In particular operations on key / value pairs can cause:

1. Out of memory errors in the driver
2. Out of memory errors on the executor nodes
3. Shuffle failures
4. "Straggler Tasks" and especially slow stages.

The first problem, memory errors in the driver, is usually caused by actions. We will discuss the performance problems associated with actions on key/value pairs in "Actions on Key/Value Pairs" on page 135. The last three performance issues - out of memory on the executors, shuffles, straggler tasks - are all most often causes by shuffles associated with the wide transformations in the `PairRDDFunctions` class. Throughout this chapter we will focus on two primary techniques to avoid performance problems associated with shuffles.

1. Shuffle less often: We will provide techniques to minimize the number of shuffles needed to achieve a complex computation. One way to minimize the number or shuffles in a computation that requires several transformations is to make sure to preserve partitioning in the narrow transformations following a wide transformation to avoid re-shuffling data (see "Preserving Partitioning Information Across Transformations" on page 151). In some instances we can use the same partitioner on a sequence of wide transformations. This can be particularly useful to avoid shuffles during joins and to reduce the number of shuffles required to compute a sequence of wide transformations (see "Co-Grouping" on page 147 and "Leveraging Co-Located and Co-Partitioned RDDs" on page 152). Relatedly, we will discuss leveraging custom partitioners (see "Custom Partitioning" on page 151) to distribute the data most effectively for down stream computations and how to push computational work into the shuffle stage to make a complicated computation more efficient (see "Secondary Sort and `repartitionAndSortWithinPartitions`" on page 153).
2. Shuffle better: Sometimes our operations will require a shuffle. However, not all wide transformations and not all shuffles are equally expensive or prone to failure. Using wide transformations such as `reduceByKey` and `aggregateByKey` that can preform map-side reductions and which do not require loading all the records for one key into memory, can prevent memory errors on the executors and speed up wide transformations (see "What's So Dangerous About the `groupByKey` Function" on page 136 and "Preventing OOM Errors with Aggregation Operations" on page 142). Lastly, shuffling with records that are distributed evenly throughout the keys, and when keys have a high number of distinct values prevents out of memory errors on the executors and straggler tasks "Straggler Detection and Unbalanced Data" on page 166.

The Goldilocks Example

Most expensive operations in Spark fit into the key/value pair paradigm because most wide transformations are key value transformations, and most require some fine tuning and care to be performant. Throughout this chapter, we will refer to a project that

the authors worked on that required finding arbitrary rank statistics in high-dimensionality and high-volume data.

The client, we will call her Goldilocks, had data of about thousands of different metrics for hundreds of millions of items. Her data looked something like this:

Table 6-1. Goldilocks example data

panda name	happiness	niceness	softness	sweetness
Mama Panda	15.0	0.25	2467.0	0.0
Papa Panda	2.0	1000	35.4	0.0
Baby Panda	10.0	2.0	50.0	0.0
Baby Panda's toy Panda	3.0	8.5	0.2	98.0

Where the attributes for each panda were doubles.

Goldilocks wanted us to design an application that would let her input an arbitrary list of integers `n1…nk` and return the `nth` best element in each column. For example, if Goldilocks input, 8, 1000, and 20 million, our function would need to return the 8th, 1000th and 20 millionth best ranking panda for each attribute column.

If we were were getting the 2nd and 4th element from the table above, we would want our function to return something like this:

Table 6-2. Goldilocks example result

column name	column index	rank statistics
happiness	1	List(3.0, 15.0)
niceness	2	List(2.0, 1000.0)
softness	3	List(35.4, 2467.0)
sweetness	4	List(0.0, 98.0).

We call this candidate "Goldilocks" because she was very picky and her house i.e. in-house cluster was crowded with other users. In this case, Goldilocks and would not accept approximate quantile boundaries, but required the output of our function to be values in the original dataset. Thus this task is inherently expensive since it will required sorting all the values in each column in some way. Lets explore how we might accomplish this task in Spark.

Because the data is columnar, we could consider Spark SQL, but early Spark SQL did not have any support for rank statistics. Although it would be possible to write a UDF/UDAF to solve the problem, the problem is complicated enough and cannot be computed on each row, so this becomes cumbersome. Thus, our solution has to leverage Spark Core.

Goldilocks Solution Version 0: Iterative Solution

One intuitive solution is to loop through each column, mapping each row to a single value, then use Spark's `sort` and `zipWithIndex` function on each column and then filter for the indices that correspond to the desired rank statistics.

For simplicity, we will assume that the columnar data was read in from stable storage as a DataFrame, that the rows are all well formed, and that the string column with the panda's name was dropped. Thus our functions takes a DataFrame of all double columns, representing the panda data, and a list of `long`s representing the positions of the elements to find for each column (e.g. 1st, 100th), and should return a map of from the column index to a list of the rank statistics in that column.

Here is an implementation of this first solution to the Goldilocks problem in which we loop through each column and sort each one using Spark's distributed sort.

Example 6-1. Goldilocks version 0, iterative solution

```
def findRankStatistics(
  dataFrame: DataFrame,
  ranks: List[Long]): Map[Int, Iterable[Double]] = {
  require(ranks.forall(_ > 0))
  val numberOfColumns = dataFrame.schema.length
  var i = 0
  var  result = Map[Int, Iterable[Double]]()

  while(i < numberOfColumns){
    val col = dataFrame.rdd.map(row => row.getDouble(i))
    val sortedCol : RDD[(Double, Long)] = col.sortBy(v => v).zipWithIndex()
    val ranksOnly = sortedCol.filter{
      //rank statistics are indexed from one. e.g. first element is 0
      case (colValue, index) =>  ranks.contains(index + 1)
    }.keys
    val list = ranksOnly.collect()
     result += (i -> list)
     i+=1
  }
```

```
    result
  }
```

This solution works, but it is slow, since it has to sort the data once for each column and does so iteratively. E.g. if we have 8,000 columns we have to do 8,000 sorts!

So how can we do better?

Since each sort can be done without knowledge of the other sorts, our intuition should be that it is possible to parallelize this computation, using each column as the unit of parallelization. We can represent the data as one long list of key/value pairs were the keys represent the column indices and perform our computation in parallel for each key.

I.e. we would map the table above to the following list of key/value pairs.

```
(1, 15.0)

(2, 0.25)

(3, 2467.0)

(4, 0.0)

(1, 2.0)

(2, 1000.0)

(3, 35.4)

(4, 0.0)

(1, 10.0)

(2, 2.0)

(3, 50.0)

(4, 0.0)

(1, 3.0)

(2, 8.5)

(3, 0.2)
```

(4, 98.0)

If we read in our data as a `DataFrame`, we can do this mapping with a simple function like the following:

Example 6-2. Goldilocks version 1, Mapping to Column Index / Value Pairs

```
def mapToKeyValuePairs(dataFrame: DataFrame): RDD[(Int, Double)] = {
  val rowLength = dataFrame.schema.length
  dataFrame.rdd.flatMap(
    row => Range(0, rowLength).map(i => (i, row.getDouble(i)))
  )
}
```

For those new to Scala, Spark's `flatMap` operation mimics the behavior of the `flatMap` operation defined on iterators and collections in Scala. `flatMap` is a very versatile narrow transformation, but it can be a bit confusing. `flatMap` lets the user define a mapping from each record to a collection of elements and then combines the resulting collections together. In this case the mapping is defined from a row to a sequence of elements `(columnIndex, value)` pairs. The resulting RDD will have more records than the previous RDD, and each will be of `(columnIndex, value)` pairs. `flatMap` can be particularly useful because unlike `map` we can return an empty collection for one of the records. Thus, the operator can be used to both filter and transform the elements in one pass. In other words a `map` and `filter` step can always be combined into one `flatMap` step.

After applying this function, we can perform this computation in parallel by column index (by key). Framed in this way, the Goldilocks problem is a key/value pair problem. Specifically:

Design a function that takes an input an RDD of integer/ double pairs and a list of longs, `n1` … `nk` and returns a map of key to a list of of k doubles that are the `n1`th, `n2`th .. `nk`th. elements for that key.

How to Use `PairRDDFunctions` and `OrderedRDDFunctions`

If you have been using Spark for a while you are probably familiar with `PairRDDFunctions` and the `OrderedRDDFunctions` class, however we provide a brief introduction

about how to use them. [1] The Spark RDD class makes use of Scala implicits, the `pairRDDFunctions` will be available on any RDD of type `(K,V)`. For pair RDD functions `K` and `V` can be of any type, but for the `OrderedRDDFunctions` (`sortByKey`, `repartitionAndSortWithinPartitions`, `filterByRange`) `K` must have some implicit ordering. Most common types like the numeric types or strings have this ordering, but in some instances you will have to define one yourself. Spark uses implicit conversion to convert an RDD that meets the PairRDD or OrderedRDD requirements from a generic type to this sub type. This implicit conversion requires that the that the correct library already be imported. So to use Spark's pairRDDFunctions you need to have imported the Spark Context. i.e imports must include:

```
import org.apache.spark.SparkContext._
```

When writing a function that uses `OrderedRDDFunctions` of generic key type, you may need to include the following line of code to establish the implicit ordering on keys. For example in the secondary sort example which we will discuss in "Secondary Sort and repartitionAndSortWithinPartitions" on page 153 we define an ordering an an object called "Panda Keys" as follows:

Example 6-3. Defining an implicit ordering to work with OrderedRDDFunctions

```
implicit def orderByLocationAndName[A <: PandaKey]: Ordering[A] = {
  Ordering.by(pandaKey => (pandaKey.city, pandaKey.zip, pandaKey.name))
}
implicit val ordering: Ordering[(K, S)] = Ordering.by(_._1)
```

Actions on Key/Value Pairs

In "Functions on RDDs: Transformations vs. Actions" on page 28 we discussed how transformations are computed on the Spark executors when an action is called. We also explained that actions usually moved data out of the Spark executors either by collecting it in the driver, or by writing to stable storage. In general we advised you to be very cautious about actions that return unbounded input to the driver as they can cause out of memory errors in the driver. However, most key/value actions (including `countByKey`, `countByValue`, `lookUp` and `collectAsMap`) return data to the driver. In most instances they return unbounded data since the number of keys and the number of values is unknown. For example, `countByKey` returns a data point for each key, thus if there are more distinct keys than fit in memory this might cause a memory

1 If you are new to these functions, Learning Spark's "Chapter 4: Working with Key/Value Pairs" provides a very good introduction.

error. Conversely, `lookUp` returns all the values for each key, so it will causes memory problems if one key has more data than will fit in memory.

The `lookUp` operation is also expensive because it triggers a shuffle if the RDD doesn't have a known partitioner.

In general, we want to try and design key/value problems so that the keys fit into memory on the driver, and the values are at least well distributed by key and at best distributed so that each key has no more records then can fit on memory on one executor. We will discuss the effects of bad key distribution in "Straggler Detection and Unbalanced Data" on page 166 and provide some suggestions for working around skewed data. As with all Spark programs if our task required bringing data back to the driver, we should try to perform transformation that reduce the size of the data before calling actions that move results to the driver.

Key/value transformations can also cause memory errors, most often in the executors, if they require all the data associated with one key to be moved kept in-memory on one partition. Avoiding memory errors and optimizing transformations for fewer shuffles is a bit more complicated than avoiding problems with actions, and so key/value transformations will be the focus of the rest of this chapter.

What's So Dangerous About the `groupByKey` Function

Many sources, including the Spark documentation, warn against the scalability of the `groupByKey` function. This section attempts to fully explain the cases in which `groupByKey` causes problems at scale and why. We also help to offer and some general ways to get around using `groupByKey`, but first I want to revisit the Goldilocks case, because my first solution to the Goldilocks problem was to use `groupByKey`.

Goldilocks Version 1: `groupByKey` solution

One, intuitive solution to the Goldilocks problem is to use `groupByKey`, which returns an iterator of each element by key. Then we can convert the iterator to another collection type, such as an array, that enables sorting. [1] After converting the iterator to an array, we can sort the array and filter for the elements that correspond to our rank statistics.

1 Iterators cannot be sorted. for more about the iterator type, its advantages and limitations, see "Iterator-to-Iterator Transformations with `mapPartitions`" on page 106.

Here is an implementation of the `groupByKey` solution. For consistency, this function also takes a DataFrame and a list of element positions. It calls the function which creates key value pairs that we described in Example 6-2.

Example 6-4. Goldilocks version 1, `GroupByKey` solution

```
def findRankStatistics(
  dataFrame: DataFrame ,
  ranks: List[Long]): Map[Int, Iterable[Double]] = {
  require(ranks.forall(_ > 0))
  //Map to column index, value pairs
  val pairRDD: RDD[(Int, Double)] = mapToKeyValuePairs(dataFrame)

  val groupColumns: RDD[(Int, Iterable[Double])] = pairRDD.groupByKey()
  groupColumns.mapValues(
    iter => {
      //convert to an array and sort
      val sortedIter = iter.toArray.sorted

      sortedIter.toIterable.zipWithIndex.flatMap({
      case (colValue, index) =>
        if (ranks.contains(index + 1))
          Iterator(colValue)
        else
          Iterator.empty
    })
  }).collectAsMap()
}

def findRankStatistics(
  pairRDD: RDD[(Int, Double)],
  ranks: List[Long]): Map[Int, Iterable[Double]] = {
  assert(ranks.forall(_ > 0))
  pairRDD.groupByKey().mapValues(iter => {
    val sortedIter  = iter.toArray.sorted
    sortedIter.zipWithIndex.flatMap(
      {
      case (colValue, index) =>
        if (ranks.contains(index + 1))
          Iterator(colValue)  //this is one of the desired rank statistics
        else
          Iterator.empty
      }
    ).toIterable //convert to more generic iterable type to match out spec
  }).collectAsMap()
}
```

This solution has several advantages. First, it is correct. Second, it is very short and easy to understand. It leverages out of the box Spark and Scala functions and so it introduces few edge cases and is relatively easy to test. On small data, it is actually

relatively efficient, because it only requires one shuffle in the `groupByKey` step and because the sorting step can be computed as a narrow transformation on the executors.

In this function, we do use `collectAsMap`, which we warned against in the previous section. In this instance, however, the danger of memory errors is minimal, because at that point of collecting, the number of keys and number of values per each key is known. The number of keys is exactly the number of columns, which we have assumed to be no larger than a few thousand, and the number of values is equal to the length of the rank statistics list we used as input for the function, which was not originally stored in a distributed way. However, it might be good practice to add a limit to the size of the input list to prevent failures in this step.

On data with 10,000 rows and a few thousand columns, this solution was much, much faster than one presented in "The Goldilocks Example" on page 130 in which I looped through the columns iteratively and sorted each one. However on a million rows the solution failed consistently without of memory exceptions even on a man node cluster.

Why `GroupByKey` Fails

If you have read `Learning Spark`, or spent much time working with Spark at scale you have probably already heard or discovered first hand that the `groupByKey` function often causes memory errors. The reason is that the "groups" created by `groupByKey` are iterators, which can't be distributed. This causes an expensive "shuffled read" step in which Spark has to read all of the shuffled data.

The following is a screen shot taken from the web ui that illustrates the high cost of `groupByKey`

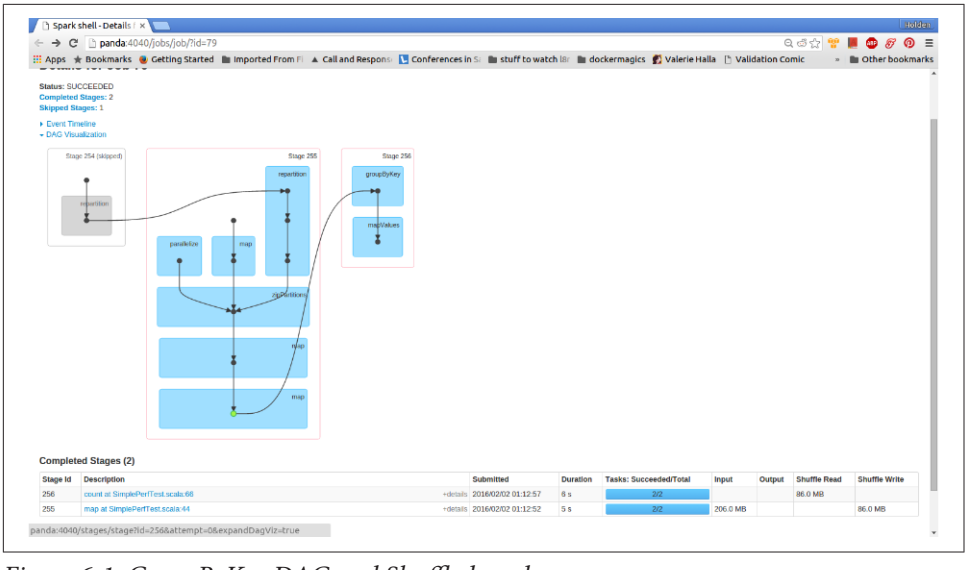


Figure 6-1. GroupByKey DAG and Shuffled read

Notice that in this computation, the shuffled read is 86 MB, although the input data is about 200 MB.

As a consequence of partitioning by the hash value of the keys, and pulling the result into memory to group as iterators, `groupByKey` often leads to out of memory error on the executors if their are many duplicate recores per key. Each record whose key has the same hash value must live in-memory on a single machine; thus, if jus one of your keys contains too many records to fit in memory on one executor, the entire operation will fail.

The following diagram illustrates a `groupByKey` operation of data about mustaches in San Fransisco. As you can see there is far more data about the 94110 zip code, which is the zip code for the Mission District, and all the records associated with that key do not fit on a single partition.

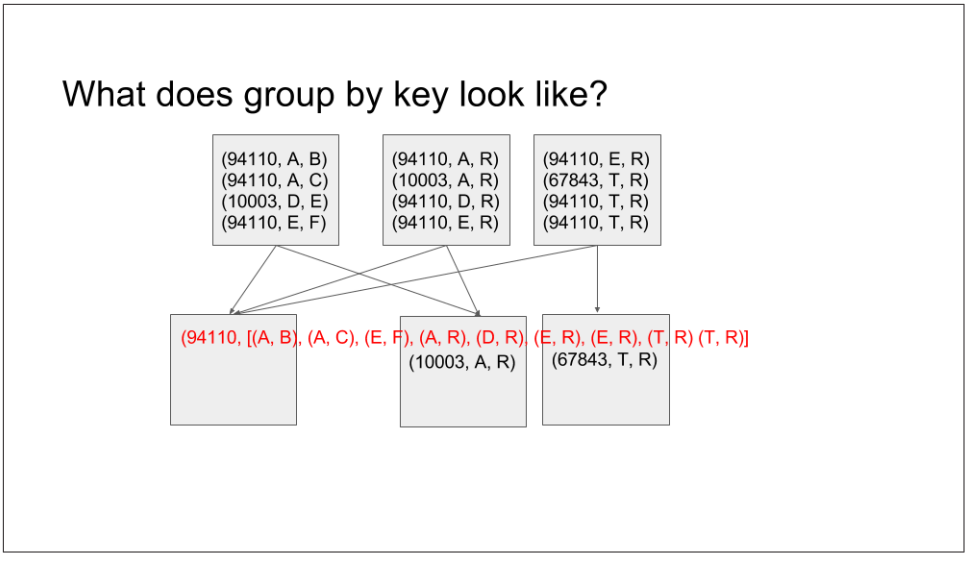


Figure 6-2. GroupByKey tip over

In general it is better to perform an operation that can do some map-side reduction to reduce the size of the data (e.g. `aggregateByKey` or `reduceByKey`) or that does not require all the values associated with one key to be kept in-memory on one machine like partitioning with a Hash Partitioner, and then grouping the values, as we discuss in "Secondary Sort and `repartitionAndSortWithinPartitions`" on page 153. If you must use `groupByKey` it is best if the next operation is an iterator to iterator transformation as discussed in "Iterator-to-Iterator Transformations with `mapPartitions`" on page 106.

Dictionary of Key/Value Operations with Performance Considerations

Many of the most important key/value pair transformations can be used to perform aggregations by key. In this section we will detail this operations and some of the performance considerations associated with them.

Aggregation Operations

Table 6-3. A dictionary of Spark's Key/Value aggregation operations

Function	Purpose	Key Restriction	Runs out of Memory When	Slow When	Output Partitioner
`groupByKey`	Group values with the same key.	Cannot have array keys with the default `HashPartitioner`. To use array keys use a custom partitioner	If just one of the keys has too many records to fit in memory on one partition	If there is not a known partitioner, this causes a shuffle. The shuffle will get more expensive as the number of distinct keys, the number of records per key increase, and if the records are not evenly distributed across the keys	`HashPartitioner` by default, but supports custom partitioning.
`combineByKey`	Combine values with the same key using a different result type.	Cannot have array keys and use the default Hash Partitioner	The "combine by" routine uses too much memory or creates too much garbage collection overhead or the accumulator for one key becomes too large (this is the problem in group by key).	Same as above.	Same as above.
`aggregateByKey`	Same as `combineByKey`, but uses one zero value for all accumulators.	See above	See above but if implemented well less likely to cause garbage collection errors (see "Minimizing Object Creation" on page 99).	See above but generally faster than `combineByKey` since it will perform the merging map-side before sending to a combiner.	Hash Partitioner by default.
`reduceByKey`	Combine values with the same key. Reduction must be to same type as original values	See above, however often less expensive then combine by key since `aggregateByKey` supports re-using the accumulator object to avoid object creation.	See above, but the type restriction make memory errors unlikely since so long as the combine is not to a collection type, the function is probably reducing. Garbage collection is less than `aggregateByKey` since no additional accumulator object is created.	Same as `aggregateByKey`	Hash Partitioner by default

Function	Purpose	Key Restriction	Runs out of Memory When	Slow When	Output Partitioner
`foldByKey`	Combine values with the same key an associative combine function, and a zero value which can be added to the result an arbitrary number of times. Use instead of `reduceByKey` when a natural `0` exists.	See above	See above	See above, performance is nearly identical to `reduceByKey`	Hash Partitioner by default.

To avoid memory allocation in `aggregateByKey`, modify the accumulator, rather than return a new one. See "Minimizing Object Creation" on page 99.

Preventing OOM Errors with Aggregation Operations

`CombineByKey` and all of the aggregation operators built on top of it (`reduceByKey`, `foldLeft`, `foldRight`, `aggregateByKey`) are no better than `groupByKey` in terms of memory errors *if* they cause the accumulator to become too large for one key. In fact, if you look up the implementation of `groupByKey`, you can see that it is actually implemented using `combineByKey` where the accumulator is an iterator with all the data, thus the accumulator is the size of all the data for that key.In other words, as long as the combining steps make the data smaller, these operations are unlikely to cause memory errors. However, if the accumulator gets larger with the addition of each new record it will eventually causes memory errors if their are many records associated with one key.

Imagine doing a back-of-the-envelope memory calculation on your sequence and combine operators:

Given the sequence operation:

```
`SeqOp(acc, v) => acc'`
```

and the combine operation

```
`combOp(acc1, acc2) = acc3.`
```

Calculate whether size(acc') < size(acc) + size(v) and size(acc3) < size(acc1) + size(acc2).

If so, the function is likely a reduction, if not, as is the case in `groupByKey`, you may need to consider a different strategy.

Beyond being less likely to run out of memory, `reduceByKey`, `treeAggregate`, `aggregateByKey` and `foldByKey`, and have free map-side combinations, meaning that records with the same key are combined before they are shuffled. This can greatly reduce the shuffled read. Compare the shuffled read in our original `groupByKey` Figure 6-1 to the amount in `reduceByKey` the following diagram. Recall that in the `groupByKey` case our shuffled read for 200mb of data ballooned to 80mb. However applying reduceByKey to the same input data reduces that number to a few hundred kilo bytes!

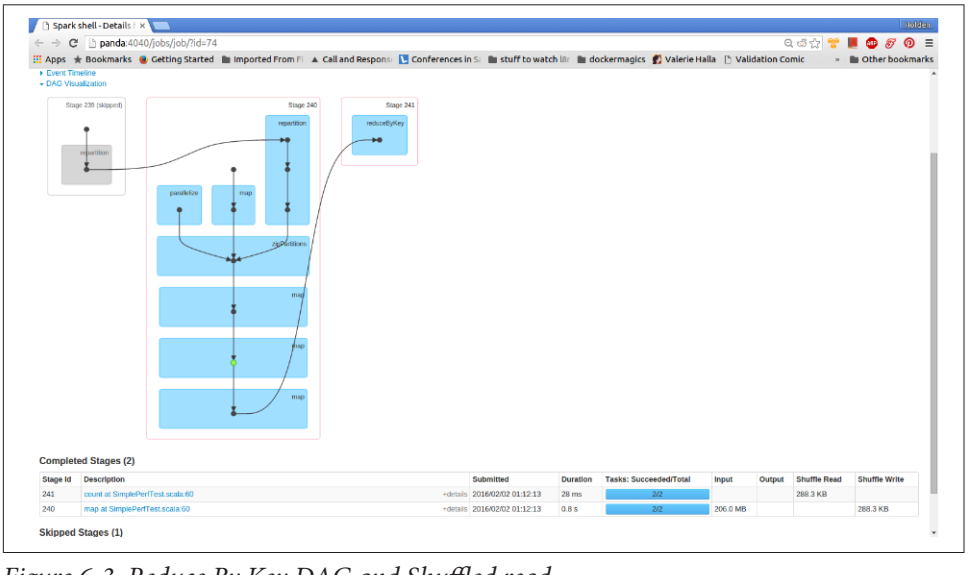


Figure 6-3. Reduce By Key DAG and Shuffled read

Other Key/Value Transformations

Table 6-4. Dictionary of operations for RDDs or Key/Value pairs

Function	Purpose	Key Restriction	Runs Out of Memory When	Slow When	Output Partitioner
`mapValues`	Apply a function to each value of a pair RDD without changing the key	none	almost never	essentially free	In contrast to `map`, this preserves the partitioning of the data for use in future operations. If you can perform your mapping on just the values it is almost always beneficial to do so.
`flatMapValues`	Apply a function that returns an iterator to each value of a pair RDD and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	none	Unlikely, unless the function applied to each value is very expensive or the result iterator is very large (causing a dramatic expansion of the number of records for each key).	essentially free	Preserves partitioning, same as the input data however, the distribution of duplicate values in the keys may be changed.
`keys`	Return an RDD of just the keys (not distinct)	none	almost never	essentially free	Preserves the partitioning of the input keys
`values`	Return an RDD of just the values	none	almost never	essentially free	Does not preserve partitioning. Future wide transformations will cause a shuffle.

Function	Purpose	Key Restriction	Runs Out of Memory When	Slow When	Output Partitioner
`sampleByKey`	Given a map with the keys to sample for and the percent of each key to take, returns a stratified sample of the input RDD. The function is implemented with `mapPartitions` and uses a random number to determine whether each record will be kept.	none	Almost none, unless the key map is too large to be broadcast to one of the worker nodes.	Same as `mapPartitions`, the function does one pass through the data and does not require a shuffle	Preserves partitioning of the input data
`partitionBy`	Takes a partitioner object and partitions the RDD accordingly. The partitioner object defines the index of the partition each record should be shuffled to based on the key	Depends on which partitioner is used. See "Using the Spark Partitioner Object" on page 149	When there are many duplicate values for each key, regardless of partitioner.	Always causes a shuffle. Range partitioners are generally slower than Hash Partitioners since they require the data to be partially evaluated in order to sample it	Partitioned according to the partition argument.

The `PairRDDFunctions` class also includes all of the multiple RDD operations such as join, which we will cover in detail in section 1.7.

OrderedRDDOperations

Table 6-5. Dictionary of operations in the OrderedRDDFunctions class

Function	Purpose	Key Restriction	Runs Out of Memory When	Slow When	Output Partitioner
`sortByKey` [a]	Return an RDD sorted by the key	Key must have implicit ordering	If data is very unevenly distributed both in terms of repeated keys and distribution of keys. In particular if the data associated with one key can't fit in memory.	Slows down with many duplicate keys. If the keys are not uniformly distributed the data won't be partitioned evenly which can also cause straggler tasks.	Range partitioner, default number of partitions is the same as the input RDD.
`RepartitionAndSortWithin Partitions`	Takes a partitioner and and an implicit ordering. Partitions the RDD according to the partitioner and then sorts all the records on each partition according to the implicit ordering	Key must have implicit ordering	Depends on the partitioner. See `partitionBy`.	Slow with key skew both because partitioner creates straggler tasks and because sorting each key will be more expensive. However this is much faster than partitioning and then sorting within each key, because the sort is pushed into the shuffle step. See "Secondary Sort and `repartitionAndSortWithin Partitions`" on page 153.	Same as partitioner argument.

Function	Purpose	Key Restriction	Runs Out of Memory When	Slow When	Output Partitioner
`filterByRange`	Takes a lower and an upper bound for the keys and returns an RDD of just the records whose key falls in that range.	See above	Almost never	If the RDD has already been partitioned by a range partitioner this is cheaper than a `filter` since it scans only the partitions whose keys are in the desired range. Otherwise performances is the same as the generic `filter` operation.	Preserves partitioning.

[a] One of the few non-pure transformations, requires an action to sample the input to create the range partitioner.

Sorting By Two Keys with `SortByKey`

Spark's `sortByKey` does allow sorting by tuples of keys. Thus it is relatively easy to sort by two values in Spark by making use of Scala's implicit tuple ordering and sort by a composite key described by `(Key, Key2)`. For example, suppose that rather than finding the `nth` item, Goldilocks just wanted us to create a directory of `(columnIndex ,value)` pairs sorted first by column index then by value. Assuming and RDD `indexValuePairs` of type `RDD[(Int, Double)]`, we could solve this problem with

```
indexValuePairs.map((_, null)).sortByKey().
```

Then `sortByKey` will use the implicit ordering on an `(Int, Double)` tuple which simply compares the first value and then compares the second.

`SortByKey` does not support implicit ordering on product types besides Tuple2. We will discuss sorting by more complex orderings in "Secondary Sort and `repartitionAndSortWithinPartitions`" on page 153.

Multiple RDD Operations

Co-Grouping

Much the same way all of the accumulator operations (`reduceByKey`, `aggergateByKey`, `foldByKey`) are implemented using `combineByKey`, all the join operations are

implemented using the `coGroup` function, which uses the `CoGroupedRDD` type. A `CoGroupedRDD` is created from a sequence of key/value RDDs each with the same key type. `CoGroup` shuffles each of the RDDs so that the items with the same value form each of the RDDs will end up on the same partition, then into a single RDD by key.

The `PairRDDFunctions` class provides several signatures for `cogroup`. `Cogroup` and its alias `groupWith`, can take one two, or three RDDs as with the same key type as arguments and regardless of value type and return an RDD with each key and then a Tuple of `Iterable` objects, where each `Iterable` is all the values of the RDDs for that key.

Suppose, for example, that we had two datasets of information about each panda, one with the scores in a series of games, and one with their favorite foods. We could use `cogroup` to associate each Panda's id with an iterator of their scores and another iterator of their favorite foods.

Example 6-5. Co-Group Example

```
val cogroupedRDD: RDD[(Long, (Iterable[Double], Iterable[String]))] = scoreRDD.cogroup(foodRDD)
```

Co-group can be useful as an alternative to join when joining with multiple RDDs. Rather than doing joins on multiple RDDs with one RDD it is more performant to co-partition the RDDs since that will prevent Spark from shuffling the RDD being repeatedly joined more than once.

For example, if we needed to join the panda score data with both address and favorite foods, it would be better to use `coGroup` than two join operations.

Example 6-6. Co-group to avoid multiple joins on the same RDD

```
val addressScoreFood = addressRDD.cogroup(scoreRDD, foodRDD)
```

However be aware that `coGroup` will cause memory errors if one key has too many records associated with it (for the same reason as `GroupByKey`). In particular `coGroup` requires that all the records in all of the co grouped RDDs for one key be able to fit on one partition. For more information on joins see Chapter 4.

Partitioners and Key/Value Data

As we covered in Chapter 2, a partition in Spark represents a unit of parallel execution which corresponds to one task. An RDD without a known partitioner will assign

data to partitions with no other considerations besides the data size and partition size [1]. The partitioner object defines a function from an element of a pair RDD to a partition via a mapping from each record to partition number. By assigning a partitioner to an RDD we can guarantee something about the data that lives in each partition, for example that it falls within a given range (range Partitioner), or includes only elements whose keys have the same hash code (Hash Partitioner).

There are three methods that exists exclusively to change the way an RDD is partitioned. In the generic RDD class `repartition` and `coalesce` can be used simply change the number of partitions that the RDD uses, irrespective of the value of the records in the RDD. As we discussed in "The Special Case of Coalesce" on page 97, `repartition` shuffles the RDD with a hash partitioner with the given number of partitions. `coalesce`, on the other hand, is an optimized version of `repartition` that avoids a full shuffle if the desired number of partitions is less than the current number of partitions. Recall that when `coalesce` reduces the number of partitions, it does so by merely combining partitions and thus is not a wide transformation since the partition can be determined at design time. When `coalesce` increases the number of partitions it has the same behavior as `repartition`. For RDDs of key/value pairs, we can use a function called `partitionBy`, which takes a partition object rather than a number of partitions and shuffles the RDD with the new partitioner. `PartitionBy` allows for much more control in the way that the records are partitioned since the partition can define a function that assigns a partition to a record based on the value of that key.

In some instances, when an RDD has a known partitioner, Spark can assume something about the data locality and avoid doing a shuffle, even if the transformation has wide dependencies. The rest of the sections in this chapter, will be about ways to leverage knowledge about partitioning or use custom partitioning, to minimize the number of times your program causes a shuffle, the distance the data has to travel in a shuffle, and the likely-hood that a disproportionate amount of the data will be sent to one partition thus (causing out of memory errors or straggler tasks).

Using the Spark Partitioner Object

At a high level, the partitioner defines how records will be distributed and consequently, which records will be completed by each task. Practically, a partitioner is actually an interface with two methods `numPartitions` and `getPartition`, which defines a mapping from a key to the integer index of the partition where records with that key should be sent. There are two implementations for the partitioner object pro-

1 Often RDDs without known partitioners can be RDDs loaded from storage, in which case the RDD's data is most often effectively partitioned in the same way as the underlying storage, but Spark does not know what the underlying partioning is so cannot take advantage of this information

vided by Spark, the `HashPartitioner` and `RangePartitioner`. If neither of these suit your needs, it is possible to define a custom partitioner.

Hash Partitioning

The default partitioner for pair RDD operations (not ordered RDD operations) is a `HashPartitioner`. A `HashPartitioner` determines the index of the child partition based on the hash value of the key. The hash partitioner requires a number of partitions parameter. If unspecified, Spark uses the value of the `spark.default.parallelism` value in the Spark Configuration to determine the number of partitions. If the default parallelism value is unset, Spark defaults to the largest number of partitions that the RDD has had in its lineage. See ??? for information and advice for setting the `spark.default.parallelism` value.

Range Partitioning

Range partitioning assigns records whose keys are in the same range to a given partition. Range partitioning is required for sorting since it ensures that by sorting records within a given partition, the entire RDD will be sorted. The range partitioner first determines the range bounds for each partition by sampling — optimizing for an equal distribution of records across partitions. Then, each record in the RDD will be shuffled to the partition whose range bounds include the key. Highly unbalanced data (i.e. lots of values for some keys and not others, and if the distribution of the keys is not uniform) make make the sampling less accurate, and, as we have discussed uneven partitioning may cause down stream tasks to be slower than others causing "straggler" tasks. If there are too many duplicate keys for all the records associated with one key to fit on one executor range partitioning, like hash partitioning, may cause memory errors. Performance problems associated with sorting are usually caused by these problems with the range partitioning step.

To create a range partitioner, Spark requires not only a number of partitions, but also the actual RDD to sample. The RDD must be a tuple and the keys must have an ordering defined.

Sampling actually requires partially evaluating the RDD, causing a break in the execution graph. Thus range partitioning is actually both a transformation and an action. The cost of sampling means that in general, range partitioning is more expensive than hash partitioning. For this reason, and also because range partitioning requires an ordering on the keys, key/value operations, such as aggregations that require records with each key to be on the same machine, but not ordered in a particular way use a Hash Partitioner as the default.

Custom Partitioning

To define a unique function for partitioning the data other than by the key's hash-value or ordering, Spark allows the user to define a custom partitioner. In order to define a partitioner you must implement the following methods:

- `numPartitions`: method which returns an integer number of partitions. Expect that this number is greater than zero.
- `getPartition` method which takes a key (of the same type as the RDD being partitioned) and returns an integer representing the index of the partition that value with that key should be sent to. The integer must be between zero and the number of partitions (defined in the `numPartitions` method)
- `equals` (optional) method to define equality between partitioners. The equality method for a HashPartitioner returns true if the number of partitions are equal. The range partitioner does so only if the range bounds are equal. The equality of partitioners can be particularly important for joins and co grouping as we will explain in "Leveraging Co-Located and Co-Partitioned RDDs" on page 152, because in some instances is an RDD is already partitioned according to a partitioner, Spark is smart enough not to shuffle again with the same partitioner.
- `hashcode` method required only if the equals method has been overridden. The hashcode of a Hash Partitioner is simply its number of partitions. The hashcode of a Range Partitioner is a hash function derived from the range bounds.

Preserving Partitioning Information Across Transformations

As we alluded to in Table 6-4 some wide transformations change the partitioning of an RDD. Spark remembers this information by updating the partitioner property of the RDD. When doing a series of transformations it is important to understand and often to preserve the information about how and RDD is partitioned to avoid doing future shuffles.

Using Narrow Transformations that Preserve Partitioning

Some narrow transformations such as `mapValues` preserve the partitioning of an RDD if it exists. Function like `map` or `flatMap` which could modify the keys according to which the RDD is partitioned will return an RDD which is not partitioned, even if the location of the records hasn't actually changed. Instead, if we don't want to modify the keys, we can call the `mapValues` function (defined only on pair RDDs) keeps the keys and therefor the partitioner exactly the same. The `mapPartitions` function will also preserve the partition if the `preservesPartitioning` flag is set to true. Assuming we have some RDD `data` of type `RDD[(Double, Int)]`, we could write the following test to illustrate this property:

Example 6-7. Maintaining partitioning information with `mapValues`

```
val sortedData = data.sortByKey()
val mapValues: RDD[(Double, String)] = sortedData.mapValues(_.toString)
assert(mapValues.partitioner.isDefined, "Using Map Values preserves partitioning")

val map = sortedData.map( pair => (pair._1, pair._2.toString))
assert(map.partitioner.isEmpty, "Using map does not preserve partitioning")
```

Leveraging Co-Located and Co-Partitioned RDDs

Co-Located RDDs are RDDs with the same partitioner that reside in the same physical location in memory. Co-location is important because all of the `CoGroupedRDD` functions (which includes the `coGroup` operations and all of the join operations) require the RDDs being grouped to have all of their partitions co-located. RDDs can only be combined without any network transfer if they have to have the same partitioner and have each of the corresponding partitions in-memory on the same executor because they were read and partitioned by the same job.

Co-partitioning is related but distinct from partition co-location. We say that multiple RDDs are *co-partitioned* if they are partitioned by the same known partitioner. We say that partitions are *co-located* if they are both loaded into memory on the same machine. RDDs are only guaranteed to be co-located if they are put into memory by the same job and the same partitioner; if one action contains the partitioning of both RDDs in its lineage. They will be Co-partitioned if the partitioner objects are equal, but not in the same physical location. Recall that "same partition" means the partition objects are equal according to the equality function defined in the partitioner class.

In the following scenario both `RDDA` and `RDDB` will be co-located:

Example 6-8. An example of co-located RDDs

```
val rddA = a.partitionBy(partitionerX)
rddA.cache()
val rddB = b.partitionBy(partitionerY)
rddB.cache()
val rddC = a.cogroup(b)
rddC.count()
```

Before Spark evaluates `RDDC.count()`, neither RDD is actually loaded into memory due to Spark's lazily evaluated nature. When Spark launches the associated `RDDC.count()` job, both RDDs are pulled into memory since their lineages are merged by the `coGroup` operation. In this case the join won't cause any network traffic because both RDDs are loaded into memory in the same location.

In contrast if we were to call:

Example 6-9. RDDs co-partitioned but not co-located

```
val rddA = a.partitionBy(partitionerX)
rddA.cache()
val rddB = b.partitionBy(partitionerY)
rddB.cache()
val rddC = a.cogroup(b)
rddA.count()
rddB.count()
rddC.count()
```

In this case, `RDDA` and `RDDB` are loaded into memory from different actions. They are co-partitioned, but there is no guarantee that there partitions will all be co-located. Thus, although the repartition calls prevent the join operator from triggering shuffles in both RDDs, there may still be some network traffic to line up the partitions and load both RDDs into memory. Although the design of your program may require calling actions in this order, it is often worth thinking about the lineage or an RDD before calling an action on it, so as to minimize network traffic.

Secondary Sort and `repartitionAndSortWithin Partitions`

Sorting in Spark could be implemented by partitioning the RDD with a `RangePartitioner` and then sorting within each partition using `mapPartitions`, much as we did in the Version 1 of the Goldilocks problem. However this approach to sorting is slower than it needs to be. Instead Spark leverages a technique called *secondary sort*, which pushes some of the work of sorting on the individual machine into the shuffle stage.

Secondary sort is a performant way of ordering data both amongst machines and within a single machine. The term comes from the MapReduce paradigm and describes a technique by which the programmer maps with one function, but defines a different order for the elements to be used in the reduce call. The effect of this in Spark is that some of the sorting work that must be done locally can be accomplished during the shuffle stage.

Spark has a built-in function to perform secondary sort called `repartitionAndSortWithinPartitions`. The `repartitionAndSortWithinPartitions` function is a wide RDD transformation that take a partitioner (defined on the argument RDD) and an implicit ordering, which much be defined on the keys of the RDD. The function partitions the data according to the partitioner argument and then sorts the records on each key according to the ordering.

We do not need to directly pass an implicit ordering to the `repartitionAndSortWithinPartitions` function. If the function is called on an RDD whose keys have an ordering, Spark can infer that ordering and will sort accordingly.To use `repartitionAndSortWithinPartitions` to order on types that either do not have an implicit ordering or have an ordering other than the one we want, we need to define the implicit ordering on the keys of the RDD before calling the `repartitionAndSortWithinPartitions` function. See Example 6-13

If we look up the implementation of `sortByKey` we can see that it calls the `repartitionAndSortWithinPartitions` function with a `RangePartitioner` and the implicit ordering defined on the keys as its arguments. As we discussed in "Range Partitioning" on page 150 the `RangePartitioner` will sample the data an assign a range of values for each partition based on the inferred distribution of the keys (for example, keys with values between 0 and 10 shall be placed on partition index two), then because `repartitionAndSortWithinPartitions` will sort the values one each partition (each range of data) the entire result will be sorted by key. The secondary sort paradigm and the `repartitionAndSortWithinPartitions` can be used not only to do a performant sort on one key, but also to define two kinds of ordering on the data, one that governs partitioning and one that governs the ordering of elements on the child partitions. The rest of this section will focus on this use case, where we want to organize the data first by one ordering and next by another.

Leveraging `repartitionAndSortWithinPartitions` for a Group By Key and Sort Values Function

The best way to order data in two ways is by using the `repartitionAndSortWithinPartitions` function directly. One common use case for this is what we might call a "group by key and sort values" use case, in which we cant to combine records with the same key and sort those records. Unlike sorting by tuple keys which we discussed in "Sorting By Two Keys with `SortByKey`" on page 147, this approach, could be generalized to any partitioning defined on the keys and any custom ordering. We use `repartitionAndSortWithinPartitions` to repartition the and use a custom ordering which will define how the records should be sorted on a given partition. The performance advantage of `repartitionAndSortWithinPartitions` is that it pushes some of the work of sorting the values within the keys into the shuffle step. In order to use `repartitionAndSortWithinPartitions` for secondary sort we need to use a custom partitioner which partitions only on the first set of keys so that the RDD will be grouped by the first key only.

With a little searching you can find numerous `groupByKeyAndSortValues` functions, although none of them have been merged into Spark. Sandy Ryza's presentation in

Advanced Analytics with Spark, is particularly good. The `groupByKeyAndSortValues` function assumes data in the form `((k, s), v)` where `s` is the secondary key (likely derived from the value) and combines the data into sorted groups of sorted values. The function has four steps

1. Define a custom partitioner that partitions records according to the value (the first element of the key)
2. Define an implicit ordering on the values. (Note, this is only necessary, because the function is generic. A tuple of primitive types will already have an implicit ordering).
3. Use `repartitionAndSortWithinPartitions` with the custom partitioner defined in step 1, on your data. Here is an simple function which performs a secondary sort on data in the form `((firsKey, secondKey), group)`.
4. Using map partitions, coalesce the items, relying on the fact that items with the same first key are grouped together, although they may not all live on the same partition, and the elements within each partition are sorted by value.

Example 6-10.

Because we are using a Hash Partitioner, this function does not actually sort values by the first key. Rather it groups those keys with the same hash value on the same machine. Thus if we have the values one through five and four partitions, the first partition will contain one and five. To force the keys to appear in sorted order we would need to define a range partitioner.

Here is an implementation of a function that sorts by the first key and then by the second:

Example 6-11. Secondary sort by sorting with two keys

```
def sortByTwoKeys[K : Ordering : ClassTag , S, V : ClassTag](
  pairRDD : RDD[((K, S), V)], partitions : Int ) = {
  val colValuePartitioner = new PrimaryKeyPartitioner[K, S](partitions)

 //tag::implicitOrdering[]
  implicit val ordering: Ordering[(K, S)] = Ordering.by(_._1)
  //end::implicitOrdering[]
  val sortedWithinParts = pairRDD.repartitionAndSortWithinPartitions(
    colValuePartitioner)
  sortedWithinParts
}
```

Here is the code for the custom partitioner:

Example 6-12. Defining a custom partitioner for Secondary sort

```
class PrimaryKeyPartitioner[K, S](partitions: Int) extends Partitioner {
  /**
   * We create a hash partitioner and use it with the first set of keys.
   */
  val delegatePartitioner = new HashPartitioner(partitions)

  override def numPartitions = delegatePartitioner.numPartitions

  /**
   * Partition according to the hash value of the first key
   */
  override def getPartition(key: Any): Int = {
    val k = key.asInstanceOf[(K, S)]
    delegatePartitioner.getPartition(k._1)
  }
}
```

We can expand this function to perform a custom grouping so that we can get a list of (secondKey, value) for each firstKey.

Example 6-13. General example of secondary sort

```
  def groupByKeyAndSortBySecondaryKey[K : Ordering : ClassTag, S, V : ClassTag]
      (pairRDD : RDD[((K, S), V)], partitions : Int
      ): RDD[(K, List[(S, V)])] = {
    //Create an instance of our custom partitioner
    val colValuePartitioner = new PrimaryKeyPartitioner[Double, Int](partitions)

    //define an implicit ordering, to order by the second key the ordering will
    //be used even though not explicitly called
    implicit val ordering: Ordering[(K, S)] = Ordering.by(_._1)

    //use repartitionAndSortWithinPartitions
    val sortedWithinParts =
      pairRDD.repartitionAndSortWithinPartitions(colValuePartitioner)

    sortedWithinParts.mapPartitions( iter => groupSorted[K, S, V](iter) )
  }

  def groupSorted[K,S,V](
    it: Iterator[((K, S), V)]): Iterator[(K, List[(S, V)])] = {
    val res = List[(K, ArrayBuffer[(S, V)])]()
    it.foldLeft(res)((list, next) => list match {
      case Nil =>
        val ((firstKey, secondKey), value) = next
        List((firstKey, ArrayBuffer((secondKey, value))))

      case head :: rest =>
        val (curKey, valueBuf) = head
```

```
        val ((firstKey, secondKey), value) = next
        if (!firstKey.equals(curKey) ) {
          (firstKey, ArrayBuffer((secondKey, value))) :: list
        } else {
          valueBuf.append((secondKey, value))
          list
        }

    }).map { case (key, buf) => (key, buf.toList) }.iterator
  }
```

How Not to Sort By Two Orderings

It's important to note that several other seemingly obvious approaches to this problem are not guaranteed to give the correct result. For example, even regardless of performance issues, `groupBykey` does not maintain the order of the values within the groups and, although if the partitioner is known it as in the following case it will not re-shuffle the data, it isn't guaranteed to maintain key order per machine, thus the following function may not give the correct results.

```
indexValuePairs.sortByKey().groupByKey()
```

Spark sorting is also not guaranteed to be stable (preserve the original order of elements with the same value) so repeated sorting is not a viable option.

```
indexValuePairs.sortByKey.map(_.swap()).sortByKey
```

Thus, the second `sortByKey` in this approach may not preserve the ordering generated in the first sort.

When developing a function like this one that relies heavily on partitioning, especially custom partitioning, make sure that your unit tests are for data that spans more than one partition. Make sure to run on different numbers of partitions and different data because there is some randomness in partitioning, especially range partitioning and the function requires assuming something about partition locality. See ??? for more about running good distributed tests.

Goldilocks Version 2: Secondary Sort

The logic of secondary sort generalizes well beyond just ordering data, and applies to any use case that requires the records to be arranged according to two different keys. The original goldilocks example is actually related to secondary sort, since it requires us to shuffle on one key (the value in the cells), and then performs an ordering by each key. Thus, rather than using `groupByKey` to insure the values associated with each key remain on each partition, and then sorting the elements associated with each key as a separate step, we can use `repartitionAndSortWithinPartitions`, par-

titioning on the column index and sorting on the value in each column. Since we know that all the values associated with each column will be on one partition and they will be in sorted order, we can simply loop through the elements on each partition, and filter for the desired rank statistics in one pass through the data.

Defining the Custom Partitioner

Because the ordering and partition in `repartitionAndSortWithinPartitions` must be defined on the keys of the RDD we need to use the (column index, value) pairs as keys. We can map to a dummy value for one or null so that Spark will interpret the RDD as key/ value pairs where the keys are a tuple of `(column index ,value)`. We will then need to define a custom partitioner that partitions the keys based on the hash value of the first part of the key (the column index) only.

Example 6-14. Goldilocks version 2, defining a custom partitioner to partition on column index

```
class ColumnIndexPartition(override val numPartitions: Int)
  extends Partitioner {
  require(numPartitions >= 0, s"Number of partitions " +
    s"($numPartitions) cannot be negative.")

  override def getPartition(key: Any): Int = {
    val k = key.asInstanceOf[(Int, Double)]
    Math.abs(k._1) % numPartitions //hashcode of column index
  }
}
```

Filtering on Each Partition

On each partition, we want the elements to be ordered first by column index, to insure that if two different column indices with the same hash value are on the same partition, the elements that are adjacent will be those with the same column indices, and then by value. Since ordering by the first value of a tuple and then the second is the existing implicit ordering on tuples, we do not have to explicitly specify an ordering for our data. Instead we can simply call `repartitionAndSortWithinPartitions` with the custom partitioner defined above. After the `repartitionAndSortWithinPartitions` call, we know that the data will be partitioned according to column index, and sorted by column index and value. For example. Suppose that we were using the DataFrame described in Table 6-1 and we were using three partitions. Then the first partition would contain the following values.

```
((1, 2.0), 1)

((1, 3.0), 1)
```

((1, 10.0), 1)

((1, 15.0),1)

((4, 0.0),1)

((4, 0.0),1)

((4, 0.0),1)

((4, 98.0),1)

Recall from our discussion of iterator-to-iterator transformations in "Iterator-to-Iterator Transformations with mapPartitions" on page 106 that the `map`, `filter`, and `flatMap` operations defined on iterators transform the elements in the iterator in order. Thus, we can use the filter operation to loop through the elements of the iterator. Since the elements are sorted and grouped by key, we can keep track of a running total for the column index and if, the element is one of the ones that corresponds to the target ranks statistic, we can keep it. We then have to map the iterator to the first half of the tuple to remove the `1` dummy value. Note, that we could combine these `map` and `filter` steps into one `flatMap` operation. I have chosen to present them separately since I think that the filter operation is easier to interpret. Like Spark, the Scala combiner combines transformations on local iterators, so this will be executed as single passes through the iterator (although in the `map` step this should be relatively few elements).

Example 6-15. Goldilocks Version 2, leveraging `repartitionAndSortWithinPartitions`

```
def findRankStatistics(dataFrame: DataFrame,
  targetRanks: List[Long], partitions: Int) = {

  val pairRDD: RDD[((Int, Double), Int)] =
    GoldilocksGroupByKey.mapToKeyValuePairs(dataFrame).map((_, 1))

  val partitioner = new ColumnIndexPartition(partitions)
   //sort by the existing implicit ordering on tuples first key, second key
  val sorted = pairRDD.repartitionAndSortWithinPartitions(partitioner)

  //filter for target ranks
  val filterForTargetIndex: RDD[(Int, Double)] =
    sorted.mapPartitions(iter => {
      var currentColumnIndex = -1
      var runningTotal = 0
      iter.filter({
        case (((colIndex, value), _)) =>
```

```
          if (colIndex != currentColumnIndex) {
            currentColumnIndex = colIndex //reset to the new column index
            runningTotal = 1
          } else {
            runningTotal += 1
          }
        //if the running total corresponds to one of the rank statistics.
        //keep this ((colIndex, value)) pair.
        targetRanks.contains(runningTotal)
    })
  }.map(_._1), preservesPartitioning = true)
  groupSorted(filterForTargetIndex.collect())
}
```

Combine the Elements Associated with One Key

After the `mapPartitions` step, we have to do one last local transformation to group the elements associated with one column index into a map. The code for the `groupSorted` function is presented below.

Example 6-16. Goldilocks version 2, group the elements associated with one key

```
private def groupSorted(
  it: Array[(Int, Double)]): Map[Int, Iterable[Double]] = {
  val res = List[(Int, ArrayBuffer[Double])]()
  it.foldLeft(res)((list, next) => list match {
    case Nil =>
      val (firstKey, value) = next
      List((firstKey, ArrayBuffer(value)))
    case head :: rest =>
      val (curKey, valueBuf) = head
      val (firstKey, value) = next
      if (!firstKey.equals(curKey)) {
        (firstKey, ArrayBuffer(value)) :: list
      } else {
        valueBuf.append(value)
        list
      }
  }).map { case (key, buf) => (key, buf.toIterable) }.toMap
}
```

Notice that this code is very similar to the grouping function presented above in Example 6-13.

Performance

This solution is considerably faster than the version 1, `groupByKey solution, on any shape of data. By using `repartitionAndSortWithinPartitions` we are able to push the work to sort each column into the shuffle stage. Since the elements are sorted after the shuffle, we are able to use an iterator-to-iterator transformations

to filter the data and avoid forcing all the values associated with one partition into memory. However, if the columns are relatively long, it may still lead to failures, since it still requires one executor to be able to store all of the values associated with all of the columns that have the same hash value. Indeed, in our case we still saw failures in the shuffle stage using this approach at scale. In fact, a viable solution to the Goldilocks problem required taking an entirely different approach.

Back to Goldilocks

Unfortunately, none of the existing key/value transformations provided a magic bullet for the Goldilocks problem. None of the other aggregations operations that we might use as alternative to `groupByKey` help us since the operation that we want to perform for each key, a sort, won't reduce the size of the data by key. As we discussed in the previous section, even re-writing our `groupByKey` approach using sophisticated secondary sort techniques was leading to failures, since in the end it still required partitioning by the column index that was not granular enough for the size of our data and the resources we had available.

Instead, a performant solution to this problem, required rethinking our approach, and how we were parallelizing it entirely, in order to play to Spark's strengths.

Before I dive into the solution, lets review some of the ways to make transformations more performant that we have learned in this chapter and the previous one.

- Narrow transformations on key/value data are quick and easy to parallelize relative to wide transformations that cause a shuffle
- Partition locality can be retained across some narrow transformations following a shuffle, such as mapPartitions if we use `preservePartitioning=true` or `mapValues`.
- Wide transformations are best with many unique keys, so that shuffles to not require a large proportion of the data to reside on one executor.
- `SortByKey` is a particularly good way to partition data and sort within partitions since it pushes the ordering of data on local machines into the shuffle stage.
- Using iterator-to-iterator transforms in `mapPartitions` prevents whole partitions from being loaded into memory
- We can sometimes rely on shuffle files to prevent re-computation of wide transformations even if call several actions on the same RDD.

Using these insights, we we were able to construct a solution to the Goldilocks problem using only one `sortByKey`, and three `mapPartitions` operations. The critical insight is that the unit of parallelization for this problem does not need to be the columns. We can essentially solve the problem for each range of values. Because, if the

cell values are sorted, and we know how many elements are on each partition from each column (that we can calculate using a performant `mapPartitions` routine) we can determine the location of the `nth` element.

Our solution can be enumerated in five steps:

1. Map the rows of data to pairs of `(cell value, index)`
2. Perform a `sortByKey` operation on all of that data
3. Using `mapPartitions`, determine how many elements in each column are on each partition and collect that information to the driver.
4. Perform a local computation on the result of step three to determine the location of each desired rank statistic. For example, suppose that we are looking for the thirteenth element and step three we determined that the first partition had ten elements from column six. Then, We can conclude that the thirteen element will be the third largest element in column 6 on the second partition.
5. Using the result of step four, do another `mapPartitions`, and filter for the elements which correspond to the desired rank statistics. Collect this information back to the driver.

Map to `(cell value, column index)` Pairs

Here is the code for step one, mapping to the `(cell value, index)` pairs. We use Spark's `flatMap` function, to transform each row into a sequence of tuples.

Example 6-17. Goldilocks algorithm version 3, map to `(cell value, column index)` pairs

```
private def getValueColumnPairs(dataFrame : DataFrame): RDD[(Double, Int)] = {
  dataFrame.rdd.flatMap{
    row: Row => row.toSeq.zipWithIndex
                .map{
                  case (v, index) => (v.toString.toDouble, index)}
  }
}
```

Sort and Count Values on Each Partition

Once we have mapped the rows so that they are keyed on cell value, we can perform the `sortByKey`. After the sort, we will have calculate the number of elements on each partition. The following is a function which takes a sorted RDD of `(double, column index)` pairs and the number of columns in the original DataFrame, and returns an array where each element corresponds to a partition. Each element of the array contains the partition index, and an array of the counts of elements on that partition each

column. The length of each sub-array will correspond to the number of columns in the original dataset.

Example 6-18. Goldilocks algorithm version 3, count values by column on each partition

```
private def getColumnsFreqPerPartition(sortedValueColumnPairs: RDD[(Double, Int)],
  numOfColumns : Int):
  Array[(Int, Array[Long])] = {

  val zero = Array.fill[Long](numOfColumns)(0)

  def aggregateColumnFrequencies (partitionIndex : Int,
    valueColumnPairs : Iterator[(Double, Int)]) = {
    val columnsFreq : Array[Long] = valueColumnPairs.aggregate(zero)(
      (a : Array[Long], v : (Double ,Int)) => {
        val (value, colIndex) = v
        //increment the cell in the zero array corresponding to this column index
        a(colIndex) = a(colIndex) + 1L
        a
      },
      (a : Array[Long], b : Array[Long]) => {
        a.zip(b).map{ case(aVal, bVal) => aVal + bVal}
      })

    Iterator((partitionIndex, columnsFreq))
  }

  sortedValueColumnPairs.mapPartitionsWithIndex(aggregateColumnFrequencies).collect()
}
```

The sub-function `aggregateColumnFrequencies` is applied to the records on each partition. It uses the aggregate operation defined on iterators. The zero value is an array the length of the original columns of zeros. For each pair in the iterator, the sequence operation of the aggregation operation increments the cell corresponding to that column index in the zero array. The combine operation adds the values in two of these array. Thus, the result is an array of the counts for the corresponding column index. For example if the first two partitions contained the following key value pairs

```
Partition 1: (1.5, 0) (1.25, 1) (2.0, 2) (5.25, 0)
Partition 2: (7.5, 1) (9.5, 2)

And there were three columns.
The output would be
[(0, [2, 1, 1]), (1, [0, 1, 1])]
```

We expect this step to be a relatively inexpensive operation. The `mapPartitions` step is a narrow transformation since it requires traversing the iterator just once, so this operation will not incur a shuffle and can spill to disk selectively. We use arrays to aggregate because as we discussed in "Using Smaller Data Structures" on page 103

they should create the least garbage collection overhead. After this `mapPartitions` step we collect the results into an array.

Example 6-19.

We are not using the result of this `mapPartitions` operation in a distributed way, since we are collecting it to the driver, thus we actually do not need to set the preserves partitioning function to false.

Determine Location of Rank Statistics on Each Partition

Once we have the results of the `getColumnsFreqPerPartition` function, we have to use that information to determine where on each partition the rank statistics are. This computation is done locally with the results of the previous function. In order to determine the location of each rank statistic we loop through the (sorted) result of the previous function keeping a running total of the elements in each column across the partitions. If for any of the columns a rank statistic is between the previous and updated value of the running total, we know that that rank statistics can be found on that partition. If this is the case we increment the `relevantIndexList` with the column index and the rank statistic - the previous running total. We can then return an array of the partition index and then a list pairs. The pairs in the list are the column index for that rank statistic and the index of that rank statics in that column. For example, if the inputs to the function were the target ranks was five, and the partition:

```
targetRanks: [5]
partitionColumnsFreq: [(0, [2, 3]), (1, [4, 1]), (2, [5, 2])]
numOfColumns: 2
```

The output will be: (0, []), (1, [(0, 3)]), (2, [(1, 1)])]

Example 6-20. Golidlocks algorithm version 3, determine location of rank statistics on each partition.

```
private def getRanksLocationsWithinEachPart(targetRanks : List[Long],
       partitionColumnsFreq : Array[(Int, Array[Long])],
       numOfColumns : Int) : Array[(Int, List[(Int, Long)])] = {

  val runningTotal = Array.fill[Long](numOfColumns)(0)
  //the partition indices are not necessarily in sorted order, so we need to sort the
  //partitionsColumnsFreq array by the partition index (the first value in the tuple)
  partitionColumnsFreq.sortBy(_._1).map { case (partitionIndex, columnsFreq) =>
    val relevantIndexList = new MutableList[(Int, Long)]()

    columnsFreq.zipWithIndex.foreach{ case (colCount, colIndex)  =>
      val runningTotalCol = runningTotal(colIndex)
      val ranksHere: List[Long] = targetRanks.filter(rank =>
        runningTotalCol < rank && runningTotalCol + colCount >= rank)
```

```
    // for each of the rank statistics present add this column index and the index it will be at
    // on this partition (the rank - the running total)
        relevantIndexList ++= ranksHere.map(rank => (colIndex, rank - runningTotalCol))

        runningTotal(colIndex) += colCount
      }

      (partitionIndex, relevantIndexList.toList)
    }
  }
```

Filter for Rank Statistics

Now, armed with the location (partition and location within each partition) of each rank statistics for each column, we have to pass through the sorted data again to filter for the correct rank statistics. This task is accomplished with the following function, called `findTargetRanksIteratively`, which uses the original sorted tuples of `(value, column index pairs)` and the results of the previous function. We use an iterator-to-iterator transformation with a filter and a map step. (Note that these could be replaced by flatMap). This produces the final result, an RDD of columnIndex, rank statistic pairs which we can then collect back to the driver.

Example 6-21. Goldilocks algorithm version 3, filter for the desired rank statistics.

```
  private def findTargetRanksIteratively(sortedValueColumnPairs : RDD[(Double, Int)],
                                         ranksLocations : Array[(Int, List[(Int, Long)])]
  ): RDD[(Int, Double)] = {

    sortedValueColumnPairs.mapPartitionsWithIndex(
      (partitionIndex : Int, valueColumnPairs : Iterator[(Double, Int)]) => {
        val targetsInThisPart: List[(Int, Long)] = ranksLocations(partitionIndex)._2
        if (targetsInThisPart.nonEmpty) {
          val columnsRelativeIndex: Map[Int, List[Long]] =
          targetsInThisPart.groupBy(_._1).mapValues(_.map(_._2))
          val columnsInThisPart = targetsInThisPart.map(_._1).distinct

          val runningTotals : mutable.HashMap[Int, Long]=  new mutable.HashMap()
          runningTotals ++= columnsInThisPart.map(columnIndex => (columnIndex, 0L)).toMap

    //filter this iterator, so that it contains only those (value, columnIndex)
    //that are the ranks statistics on this partition
    //Keep track of the number of elements we have seen for each columnIndex using the
    //running total hashMap.
          valueColumnPairs.filter{
            case(value, colIndex) =>
              lazy val thisPairIsTheRankStatistic: Boolean = {
                val total = runningTotals(colIndex) + 1L
                runningTotals.update(colIndex, total)
                columnsRelativeIndex(colIndex).contains(total)
              }
```

```
            (runningTotals contains colIndex) && thisPairIsTheRankStatistic
        }.map(_.swap)
      }
      else {
        Iterator.empty
      }
    })
  }
```

Goldilocks Version 3: Sort on Cell Values

Combining all of these functions together we get a full solution to the goldilocks problem.

Example 6-22. Goldilocks Sort On Values

```
def findRankStatistics(dataFrame: DataFrame, targetRanks: List[Long]):
  Map[Int, Iterable[Double]] = {

  val valueColumnPairs: RDD[(Double, Int)] = getValueColumnPairs(dataFrame)
  val sortedValueColumnPairs = valueColumnPairs.sortByKey()
  sortedValueColumnPairs.persist(StorageLevel.MEMORY_AND_DISK)

  val numOfColumns = dataFrame.schema.length
  val partitionColumnsFreq =
    getColumnsFreqPerPartition(sortedValueColumnPairs, numOfColumns)
  val ranksLocations =
    getRanksLocationsWithinEachPart(targetRanks, partitionColumnsFreq, numOfColumns)

  val targetRanksValues = findTargetRanksIteratively(sortedValueColumnPairs, ranksLocations)
  targetRanksValues.groupByKey().collectAsMap()
}
```

From a code readability perspective, this solution is ugly. It is many steps, four passes through the data, and requires throwing away information. However, we expect that it will avoid memory errors on the executors and complete faster than the `groupByKey` or secondary sort solutions, since the data on each column should be mostly distinct doubles, and thus the shuffle should be fairly efficient. The final two `mapPartitions` routines involve reducing the data and can be achieved through iterator-to-iterator transformations, so we expect them to scale well. On well balanced data with many records this solution out performs the previous ones.

Straggler Detection and Unbalanced Data

"Stragglers" are those tasks within a stage which take much longer to execute than the other tasks in one stage. Recall from our discussion in "Spark Job Scheduling" on page 31, that a new stage begins after each wide transformation, and usually, when a

wide transformations are called on the same RDD, stages must be executed in sequence, so straggler tasks may hold up an entire job. Stragglers occur when Spark has not allocated resources correctly, in particular if the data has not been partitioned evenly. Stragglers are a good indication of unbalanced keys, since in a shuffle operation distribution of tasks depends on partitioning, which in turn depends on the keys. The Spark Web UI allows you to monitor tasks as they are executed in real time.

If during a wide transformation you notice that some partitions take much longer than others, and have more failed tasks its likely that the data is not being partitioned evenly, likely because some keys have many more values than others. In this case it will speed up shuffle operations to either use something else as keys, or add random "noise" to you keys to create more distinct keys, perform a map side reduction to combine or filter records with duplicate on each node before shuffling all the data.

While `sortByKey` is less likely to cause memory errors at scale than `groupByKey`, it is still quite possible. Think back to Figure 6-2 and see how we can still tip over with a sortByKey in Figure 6-4.

One work around to unbalanced keys, can be adding "junk" to the end of the key such as a random number, that way Spark can recognize the keys as distinct and spread them across partitions. In our case, this could mean spreading the zeros across machines, which should not affect the accuracy. The following images are meant to illustrate an imbalanced `sortByKey`, Figure 6-4, and a balanced `sortByKey`, Figure 6-5.

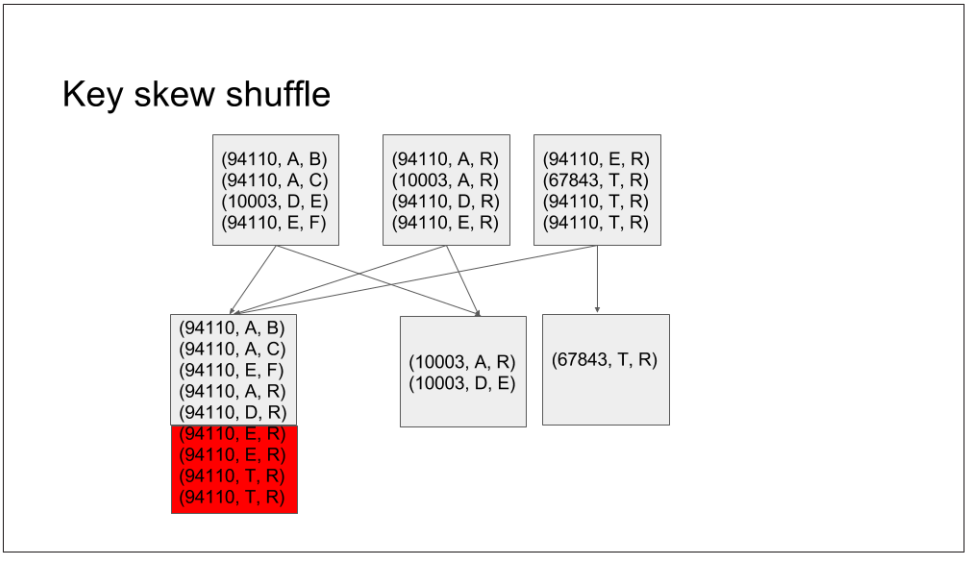


Figure 6-4. SortByKey Memory Errors

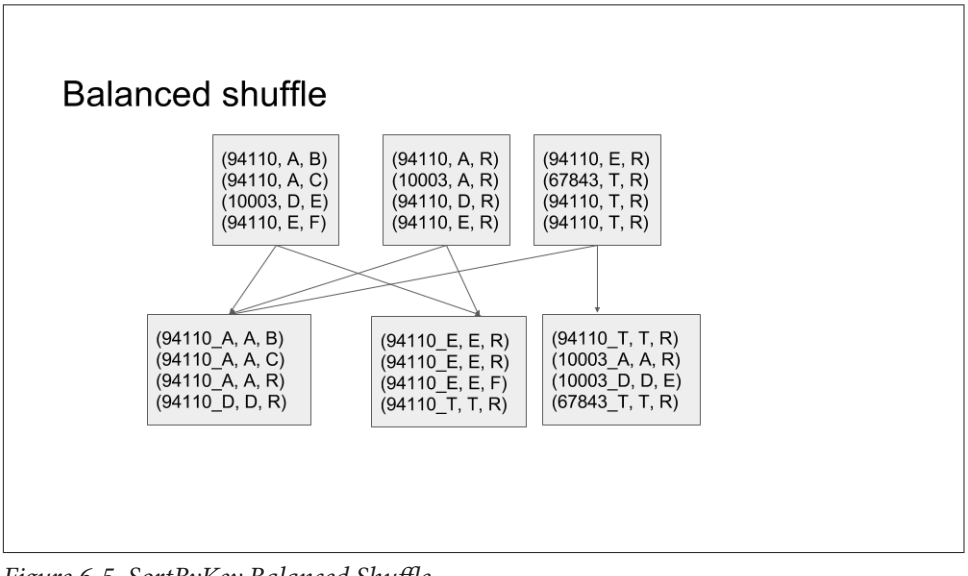


Figure 6-5. SortByKey Balanced Shuffle

Back to Goldilocks (Again)

Goldilocks has told us that her data was evenly distributed and that all the columns had "lots" of distinct values. However, when we were running our initial rank-statistics algorithm we noticed that some partitions were running much slower than

the others, and sometimes running out of memory. This indicated that we had too many duplicate keys. Since in our implementation, the keys were the values of the data (doubles) this was a bit surprising. However, further analysis revealed that in most of the columns, about twenty-five percent of the data was zeros. Thus, nearly one in four keys in our sort was a zero value. This mean that 1/4 of the dat was clustered around zero and was causing the first few partitions to get far more data than the other ones if we were running with more than four partitions.

Goldilocks Version 4: Reduce to Distinct on Each Partition

Rather than trying to partition those pairs differently, we realized that last four steps of the algorithm could be modified to work on input of tuples of `((cell value, column index), count)`. Then, in the first step, rather than mapping the records on each partition to `(cell value, column index)` pairs, we could map each partition to distinct pairs, keeping track of the number of times each `((cell value, column index))` pair appeared on that partition. By mapping to distinct on each partition, we reduce the number of duplicate keys without incurring a shuffle (as doing a full reduce to distinct would).

After creating these `(value, columnIndex), count)` tuples, we know that although we might have some repeated keys, we introduce a theoretical limit on the number of duplicate keys. Specifically, if the same value is present in a column on each partition, the maximum number of duplicate keys is (the number of columns * the number of partitions) [1]. Not only does this input step balance the data so it can be partitioned more effectively, it also reduces the total number of records to be shuffled dramatically. In the Goldilocks case, where 25 percent of the rows were zero, we now have roughly 75% the number of tuples to sort as in our version three of this algorithm. We found that this optimization lead to a 4-5x speed up on a large cluster and was required for the job to complete on a small, noisy cluster at all.

Here is the code for this first step, rather than mapping to `(cell_value, column_index)` pairs, we mapped to `((cell value, column index), count)` on each partition. We were getting the key/value pairs using a map partitions step. Now, rather than doing a flat map of each row in the original data, we update a `HashMap` whose keys are the `(value, index)` pairs and whose values are the number of times that `(cell value, column index)` pair appears in that partition. Even if each column on a given partition has all unique values, we expect that this solution will not cause out of memory errors since we are making the `HashMap` on each partition, and so the size of the `HashMap` should be smaller than the iterator of all the records. However, note that if, in fact we had all distinct values, this might not be the case since as

1 using reduce by could would have reduced this to simply the number of columns

we have discussed, Hash maps are a much less memory efficient data structure than iterators. won't be any larger than the size of the data that was on that partition before the `mapPartitions` step.

Aggregate to `((cell value, column index), count)` on Each Partition

Example 6-23. Goldilocks version 4, aggregate on each partition

```
def getAggregatedValueColumnPairs(dataFrame : DataFrame) : RDD[((Double, Int), Long)] = {

  val aggregatedValueColumnRDD =  dataFrame.rdd.mapPartitions(rows => {
    val valueColumnMap = new mutable.HashMap[(Double, Int), Long]()
    rows.foreach(row => {
      row.toSeq.zipWithIndex.foreach{ case (value, columnIndex) =>
        val key = (value.toString.toDouble, columnIndex)
        val count = valueColumnMap.getOrElseUpdate(key, 0)
        valueColumnMap.update(key, count + 1)
      }
    })

    valueColumnMap.toIterator
  })

  aggregatedValueColumnRDD
}
```

Sort and Find Rank Statistics

The rest of the function is similar to the original version, we just adjust for keeping track of the number of times the pair appears, rather than assuming each pair in the sorted RDD occurred once.

Example 6-24. Goldilocks version 4, count values per column on each partition

```
private def getColumnsFreqPerPartition(
  sortedAggregatedValueColumnPairs: RDD[((Double, Int), Long)],
  numOfColumns : Int): Array[(Int, Array[Long])] = {

  val zero = Array.fill[Long](numOfColumns)(0)

  def aggregateColumnFrequencies(
    partitionIndex : Int, pairs : Iterator[((Double, Int), Long)]) = {
    val columnsFreq : Array[Long] = pairs.aggregate(zero)(
      (a : Array[Long], v : ((Double,Int), Long)) => {
        val ((value, colIndex), count) = v
        a(colIndex) = a(colIndex) + count
        a},
      (a : Array[Long], b : Array[Long]) => {
        a.zip(b).map{ case(aVal, bVal) => aVal + bVal}
      })
```

```
    Iterator((partitionIndex, columnsFreq))
  }

  sortedAggregatedValueColumnPairs.mapPartitionsWithIndex(aggregateColumnFrequencies).collect()
}
```

Example 6-25. Goldilocks version 4, determine locations of rank statistics on each partition

```
private def getRanksLocationsWithinEachPart(targetRanks : List[Long],
       partitionColumnsFreq : Array[(Int, Array[Long])],
       numOfColumns : Int) : Array[(Int, List[(Int, Long)])]  = {

  val runningTotal = Array.fill[Long](numOfColumns)(0)

  partitionColumnsFreq.sortBy(_._1).map { case (partitionIndex, columnsFreq)=>
    val relevantIndexList = new mutable.MutableList[(Int, Long)]()

    columnsFreq.zipWithIndex.foreach{ case (colCount, colIndex)  =>
      val runningTotalCol = runningTotal(colIndex)

      val ranksHere: List[Long] = targetRanks.filter(rank =>
        runningTotalCol < rank && runningTotalCol + colCount >= rank)
      relevantIndexList ++= ranksHere.map(rank => (colIndex, rank - runningTotalCol))

      runningTotal(colIndex) += colCount
    }

    (partitionIndex, relevantIndexList.toList)
  }
}
```

Example 6-26. Goldilocks version 4, filter for rank statistics

```
private def findTargetRanksIteratively(
        sortedAggregatedValueColumnPairs : RDD[((Double, Int), Long)],
        ranksLocations : Array[(Int, List[(Int, Long)])]): RDD[(Int, Double)] = {

  sortedAggregatedValueColumnPairs.mapPartitionsWithIndex((partitionIndex : Int,
    aggregatedValueColumnPairs : Iterator[((Double, Int), Long)]) => {

    val targetsInThisPart: List[(Int, Long)] = ranksLocations(partitionIndex)._2
   if (targetsInThisPart.nonEmpty) {
      FindTargetsSubRoutine.asIteratorToIteratorTransformation(aggregatedValueColumnPairs,
        targetsInThisPart)
    }
    else Iterator.empty
  })
}
```

Example 6-27. Goldilocks version 4, iterator-to-iterator transformation to filter for the rank statistics

```
  def asIteratorToIteratorTransformation(valueColumnPairsIter : Iterator[((Double, Int), Long)],
    targetsInThisPart: List[(Int, Long)] ): Iterator[(Int, Double)] = {

    val columnsRelativeIndex = targetsInThisPart.groupBy(_._1).mapValues(_.map(_._2))
    val columnsInThisPart = targetsInThisPart.map(_._1).distinct

    val runningTotals : mutable.HashMap[Int, Long]=  new mutable.HashMap()
     runningTotals ++= columnsInThisPart.map(columnIndex => (columnIndex, 0L)).toMap

    //filter out the pairs that don't have a column index that is in this part
    val pairsWithRanksInThisPart =  valueColumnPairsIter.filter{
      case (((value, colIndex), count)) =>
        columnsInThisPart contains colIndex
     }

    //map the valueColumn pairs to a list of (colIndex, value) pairs that correspond to one of the
    //desired rank statistics on this partition.
    pairsWithRanksInThisPart.flatMap{

      case (((value, colIndex), count)) =>

          val total = runningTotals(colIndex)
          val ranksPresent: List[Long] = columnsRelativeIndex(colIndex)
                                        .filter(index => (index <= count + total)
                                          && (index > total))

          val nextElems: Iterator[(Int, Double)] =
            ranksPresent.map(r => (colIndex, value)).toIterator

          //update the running totals
          runningTotals.update(colIndex, total + count)
          nextElems
    }
  }
```

Putting all the code together we have a final solution to the Goldilocks problem

Example 6-28. Goldilocks with hash map

```
  def findRankStatistics(dataFrame: DataFrame, targetRanks: List[Long]):
    Map[Int, Iterable[Double]] = {

    val aggregatedValueColumnPairs: RDD[((Double, Int), Long)] =
      getAggregatedValueColumnPairs(dataFrame)
    val sortedAggregatedValueColumnPairs = aggregatedValueColumnPairs.sortByKey()
    sortedAggregatedValueColumnPairs.persist(StorageLevel.MEMORY_AND_DISK)

    val numOfColumns =  dataFrame.schema.length
    val partitionColumnsFreq =
```

```
    getColumnsFreqPerPartition(sortedAggregatedValueColumnPairs, numOfColumns)
  val ranksLocations  =
    getRanksLocationsWithinEachPart(targetRanks, partitionColumnsFreq, numOfColumns)

  val targetRanksValues =
    findTargetRanksIteratively(sortedAggregatedValueColumnPairs, ranksLocations)
  targetRanksValues.groupByKey().collectAsMap()
}
```

Conclusion

In this chapter, we have seen how to use functions in the `PairRDDFunctions` and `OrderedRDDFunctions` classes in ways that are more likely to succeed at scale. In particular, we have learned to be cautious about aggregation operations which don't reduce the space needed to store all the records associated with each key, such as `groupByKey`. Key/value operations often cause shuffles. We have learned about partitioning and how thinking ahead towards the next key/value transformation and doing smart partitioning can reduce the number of shuffles we have to do in our program. We have focused on some strategies to do fewer shuffles - by using smart partitioning, maintaining partition information with narrow transformations, leveraging co-location for joins. When shuffles are required, we learned that some distributions of data are more likely to cause failures in the shuffle stage than others. We have shown that unbalanced data, particularly a high number of duplicate values per key, is likely to slow down shuffles and cause memory errors, and have provided some examples of how to clean data to avoid these problems.